UNIVERSIDAD CARLOS III DE MADRID

Escuela Politécnica Superior

GRADO EN INGENIERÍA ELECTRÓNICA INDUSTRIAL Y AUTOMÁTICA

DEPARTAMENTO DE SISTEMAS Y AUTOMÁTICA

TRABAJO FIN DE GRADO

Mapeado del Entorno mediante Visión Estéreo

Autor: Ángel Madridano Carrasco

Tutor: Pablo Marín Plaza

Junio 2015

Dedicado a mis padres, hermanas y pareja por haberme apoyado día tras día en estos cuatro años de carrera, y por hacer posible este proyecto…

Dar las gracias al departamento de Sistemas y Automática, en especial a J.M. Armingol, a F. García y a mi tutor P. Marín, por el apoyo y confianza depositada en mí para desarrollar el presente proyecto.

ÍNDICE

ÍNDICE DE FIGURAS

ABREVIATURAS

OMS	Organización Mundial de la Salud
iCab	Intelligent Campus Automobile
ROS	Robot Operating System
ADAS	Advanced Driver Assistance Systems (Sistemas Avanzados de Asistencia al Conductor)
ITS	Intelligent Transport System (Sistemas Inteligentes de Transporte)
DARPA	Defense Advanced Research Projects Agency (Agencia de Investigación de Proyectos Avanzados de Defensa)
SLAM	Simultaneous Localization and Mapping
LSI	Laboratorio de Sistemas Inteligentes
IvvI	Intelligent Vehicle based on Visual Information (Vehículo Inteligente basados en Información Visual)

1. RESUMEN

La industria del automóvil posee un papel fundamental en la economía y desarrollo de las sociedades y los países que contrasta con un elevado número de personas, que según la OMS, fallecen y resultan heridas en las carreteras del mundo debido a accidentes de tráfico cuya primera causa, según estudios realizados en los últimos años, son los factores humanos. De ahí que la idea de emplear vehículos autónomos posea cada vez más adeptos.

Para la implantación de este transporte totalmente autónomo es necesaria una ley, que regule tanto la circulación como los permisos y características de uso de este tipo de vehículos. En la actualidad, pocos países poseen una legislación que permita el tráfico de coches autónomos, por lo que muchos de los centros de investigación en tecnologías relacionadas con la autonomía de los medios de transporte emplean sistemas de transporte inteligentes que pueden actuar en entornos urbanos para desarrollar y probar nuevos avances tecnológicos.

Desde el punto de vista tecnológico, los vehículos autónomos deben estar provistos de sistemas de navegación con sensores y dispositivos capaces de visualizar el entorno y generar, a través de los datos recogidos, un mapa que recoja los obstáculos y espacio libre presentes en el horizonte, para garantizar una navegación segura del vehículo.

En este proyecto se implementarán una serie de algoritmos para el sistema inteligente de transporte iCab que permitan obtener, haciendo uso de la plataforma de software ROS, un mapa del entorno a través de la información captada desde un sistema de visión estéreo.

El proyecto iCab está siendo implementado por el grupo de investigación LSI (Laboratorio de Sistemas Inteligentes) de la Universidad Carlos III de Madrid, con el objetivo de desarrollar vehículos autónomos que puedan desenvolverse por el Campus de Leganés sin necesitar necesidad de interacción humana.

En este proyecto se desarrolla un algoritmo que permite generar un Grid Map gracias a la información aportada por una cámara estéreo.

Palabras clave:

Vehículos autónomos, sistemas de navegación, ROS, visión estéreo, obstáculos.

ABSTRACT

The automotive industry plays a key role in the economy and development of societies and countries. By contrast, according to the WHO, a large number of people die or are injured in traffic accidents on roads around the world, and the first cause is the human factor, according to studies in recent years. For this reason, the idea of using driverless vehicles is increasingly popular.

To implement this fully driverless transport is needed a law to regulate both the circulation and permits and usage characteristics of this type of vehicle. Currently, few countries have legislation that allows the driverless car in real-life traffic conditions, so many of the centers investigating driverless transport technologies use intelligent transport systems in urban areas in order to develop and test new technological advances.

From the technological point of view, driverless vehicles must be equipped with navigation systems with sensors and devices capable of distinguishing the environment and generate, from collected data, a map showing the obstacles and free space on the horizon to ensure safe driving performance.

In this project, a series of algorithms will be implemented for the intelligent transport system iCab, using ROS software platform to generate a map of the environment based on the information captured by a stereo vision system.

The iCab project is being implemented by the research group LSI (Laboratory of Intelligent Systems) of the Carlos III University of Madrid, with the aim of developing autonomous vehicles that could manage on the campus of Leganes without requiring the need for human interaction.

In this project an algorithm to generate a Grid Map thanks to the information provided by a stereo camera is developed.

Keywords:

Autonomous vehicles, navigation systems, ROS, stereo vision, obstacles.

2. INTRODUCCIÓN

2.1. MOTIVACIÓN

La industria automotriz se establece como uno de los principales motores económicos del mundo con un mercado global que se cifra actualmente en 2.000 millones de dólares y, los automóviles como el medio de transporte, tanto de pasajeros como de mercancías, más empleado en la Unión Europea [1]. En contrapartida a estos datos positivos se tiene que los accidentes de tráficos suponen la novena causa de muerte en el mundo (Fig. 2.1), con 1.3 millones de muertos al año, y con una previsión de alcanzar los 2 millones de muertos en el año 2030 [2]. En la actualidad, el 90% de los accidente de tráfico son causados por el factor humano, entre los que destacan las colisiones con objetos que el conductor no ha visto, bien por distracción o por falta de visibilidad.

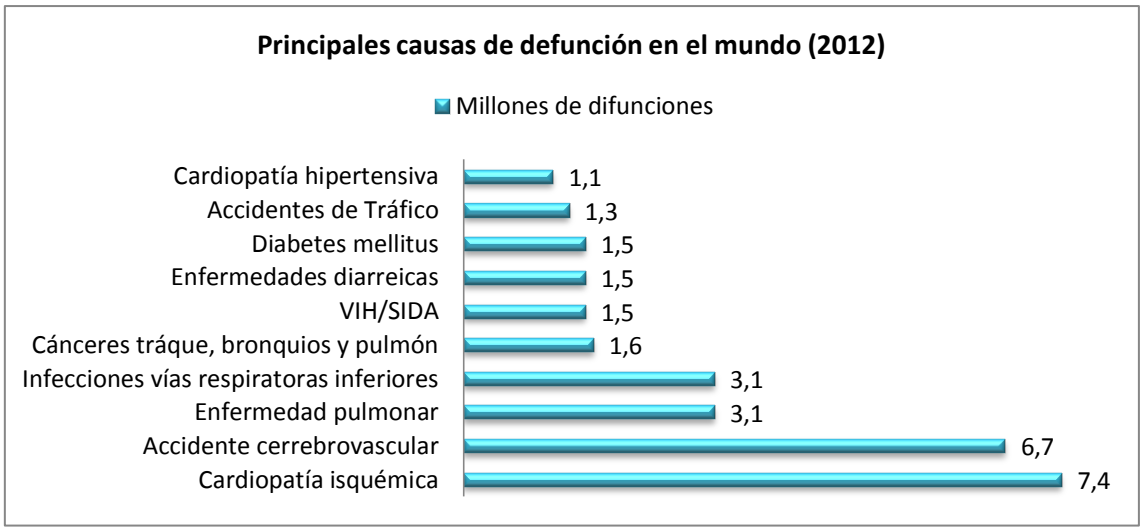


Fig. 2.1. Causas de muerte en el mundo en millones de defunciones

Combatir estos elevados datos de siniestralidad fue uno de los principales motivos del desarrollo de los vehículos autónomos y los sistemas inteligentes de transportes. Existen datos que dan pie al optimismo de este sector, pues se prevé que en 2046 todos los coches serán autónomos, y que ya en 2026 se empezarán a ver numerosos ejemplos de coches con sistema de conducción pilotada o autónoma en los que se haya perfeccionado el complejo sistema de este tipo de automóviles y se haya llevado a cabo una notable mejora de las infraestructuras [3].

Actualmente, los fabricantes de automóviles, están incorporando a sus modelos comercializados gran cantidad de Sistemas Avanzados de Asistencia al Conductor o ADAS, que van desde la monitorización del conductor para advertir de descensos de su atención en la conducción hasta sistemas *ecall* capaces de producir llamadas de socorro, proporcionando datos del vehículo y su estado, a los centros de emergencia en caso de accidente. El avance y desarrollo de estas tecnologías está desembocando, junto con las mejoras de las carreteras e infraestructuras, en la implantación a nivel mundial de Sistemas Inteligentes de Transporte, los cuales poseen como uno de sus objetivos el desarrollo de vehículos autónomos. Los ADAS se podrían catalogar como la aplicación de los ITS en los vehículos.

Los vehículos autónomos son capaces de imitar las capacidades humanas de manejo y control y, realizar todas las tareas necesarias para moverse de un punto a otro sin ninguna intervención humana, para lo cual poseen un conjunto de sensores y sistemas que le permiten percibir el medio que le rodea y navegar en consecuencia.

2.1.1. OBJETIVOS

El presente proyecto tiene como principal objetivo crear, a través de la información obtenida de una cámara binocular, un mapa del entorno próximo al vehículo autónomo iCab.

La generación de un mapeado del entorno requiere superar un conjunto de tareas secundarias cuya resolución permitirá lograr la meta principal que concierna a este trabajo, y que son:

- Transformar el mapa de disparidad, generado a partir de las imágenes rectificadas captada por la cámara binocular, en una nube de puntos.
- Crear, con el conjunto de datos obtenidos de la nube de puntos, un Fake laser o escaneo láser virtual que permita rellenar las cuadrículas de un Grid Map con los obstáculos presentes en el entorno.

2.2. SISTEMAS INTELIGENTES DE TRANSPORTE

El crecimiento de los parques de automóviles durante la segunda mitad del siglo XX trajo consigo una serie de objetivos: el aumento y mejora de la seguridad, la optimización de la red de carreteras y la reducción del consumo energético y de los niveles de contaminación. En respuesta a estas necesidades surgió la idea de la conducción automática. Con este tipo de conducción se busca alcanzar diversas metas como la detección de las líneas de carril, el mantenimiento de la distancia de seguridad, la regulación de la velocidad según las condiciones de tráfico y el tipo de vía, la optimización de la ruta y la ayuda a la circulación y aparcamiento en entornos urbanos.

Así nacieron, hace unos 35 años, los Sistemas de Transporte Inteligentes o ITS (*ITS ≡Intelligent Transport System)* que se centran en mejorar la seguridad, eficiencia y confort de cualquier medio de transporte. Los ITS (Fig. 2.2) establecen una interfaz entre el vehículo y la carretera con objetivo de facilitar la conducción, sin realizar cambios físicos en las infraestructuras ya existentes. Una de las metas a alcanzar con los ITS es el desarrollo de vehículos autónomos, es decir, conseguir obtener un conductor artificial capaz de manejar los dispositivos del vehículo tal y como lo haría un conductor humano pero con mayor grado de seguridad y eficacia [4].

Fig. 2.2. Sistemas Inteligentes de Transporte

La historia de los coches autónomos nace en la Exposición Universal de 1939 con la presentación de un vehículo eléctrico, diseñado por Norman Bel Geddes, que era controlado por un circuito eléctrico embebido en el pavimento de la carretera [5]. En las siguientes décadas fue aumentando la investigación y se fueron sucediendo las apariciones de distintos vehículos capaces de desplazarse sin intervención humana. Los principales avances llegaron desde la universidad estadunidense Carnegie Mellon, que en 1995 desarrolló el Navlab 5 (Fig. 2.3) capaz de circular de forma autónoma un 98.2% en 5000 km, y, las universidades europeas de Múnich, capaz de crear en 1980 una furgoneta que alcanzará sin ayuda humana los 100 km/h en una autopista sin conductor, y Parma, que en 1996 consigue que su vehículo ARGO recorra 2000 km por autopistas del norte de Italia a una velocidad media de 90 Km/h y con un 94 % del tiempo en modo de conducción automática[6].

Fig. 2.3. Vehículo Navlab 5 de Carnegie Mellon

El interés de la Agencia de Investigación de Proyectos Avanzados de Defensa (DARPA), principal organismo de investigación del Departamento de Defensa de los Estados Unidos, impulsó la investigación en este campo, creando en 2004 el DARPA Grand Challenge, una competición de vehículos autónomos con un premio de un millón dólares para el ganador de la prueba, con la que buscaban atraer investigadores para alcanzar el objetivo de automatizar un tercio de los transportes militares en 2015.

La primera DARPA Grand Challenge se celebró el 13 de marzo de 2004 en el desierto de Mojave, con una ruta de 150 millas que unía Barstow (California) con Primm (Nevada), y en la que ninguno de los 15 finalistas consiguió completar la prueba. El ganador fue el vehículo Sandstorm de la Carnegie Mellon (Fig. 2.4) que recorrió 7.5 millas antes de abandonar.

Fig. 2.4. Vehículo Sandstorm ganador del DARPA Grand Challenge 2004

El 8 de octubre de 2005 se celebró de nuevo la DARPA Grand Challenge, con un recorrido de 131.6 millas, en esta ocasión hubo 23 finalistas, de los cuales 5 vehículos llegaron a la meta, y cuatro lo hicieron por debajo del límite de tiempo propuesto. En esta ocasión el ganador fue el vehículo Stanley (Fig.2.5), desarrollado en la Universidad de Stanford, en un proyecto dirigido por Sebastian Thrun. El vehículo era un Volkswagen Touareg que realizó el trayecto en 6 horas y 54 minutos provisto de cinco sensores láser para la detección de obstáculos y cámaras de video a color para distancias de 50 metros.

Fig. 2.5. Vehículo Stanley ganador de la DARPA Grand Challenge 2005

En el 2007 se celebró la tercera y última competición hasta el momento de la Grand Challenge. Se denominó Urban Challenge, y los vehículos tenían que realizar el recorrido obedeciendo a las reglas de tráfico e interactuando unos con otros. Hubo 11 finalistas, siendo el ganador el prototipo Tartan Racing de la Universidad Carnegie Mellon [6].

En octubre del año 2010 Google informó que poseía vehículos autónomos desarrollados por Sebastian Thrun y Chris Urmson, de las universidades de Stanford y Carnegie Mellon, capaces de circular sin intervención del conductor empleando la técnica de navegación SLAM (Simultaneous Localization and Mapping), en la que partiendo de un mapa del entorno previo, el vehículo lo construye y lo actualiza. El desarrollo de la legislación y las mejoras en los sistemas han llevado a Chris Urmson a afirmar que el coche autónomo está listo para ser probado en carreteras reales, y que en el verano de 2015 será posible ver los primeros coches de Google por las calles de California, cumpliendo una serie de requisitos [7]. Los prototipos que recorrerán las carreteras deberán circular con una velocidad máxima de 40 Km/h para evitar posibles accidentes, y lo harán bajo la supervisión de un operador humano que podrá tomar el control del vehículo en caso de que sea necesario. Estos prototipos que han recorrido 16.000 kilómetros de pruebas, lo que les hace tener una experiencia equivalente a la de un conductor que lleva 75 años al volante, buscan con esta fase conocer cómo la comunidad recibe e interactúa con estos vehículos, para así mejorar su funcionamiento y rendimiento.

2.3. PLATAFORMAS DE INVESTIGACIÓN

La Universidad Carlos III de Madrid cuenta con un Laboratorio de Sistemas Inteligentes (LSI) con dilatada experiencia en desarrollar soluciones innovadoras en el campo de la automatización, control y optimización de sistemas de fabricación de distintos sectores. De entre las diversas líneas de investigación en las que trabaja este laboratorio, dos de ellas están ampliamente relacionadas con el presente proyecto, como son los sistemas inteligentes de transporte y la visión por computador. En el área de los ITS la universidad posee distintos modelos de vehículos donde poder testar las implementaciones desarrolladas en dicho campo.

2.3.1. IvvI / IvvI 2.0

El LSI lleva varios años trabajando en el Vehículo Inteligente basado en Información Visual (IvvI). En la actualidad los sensores y equipos necesarios para el desarrollo de sistemas de ayuda a la conducción se encuentran embarcados en un Nissan Note, segundo vehículo empleado para el estudio y desarrollo de estas tecnologías.

Fig. 2.6. Plataforma de Investigación Ivvl A) Vehículo Renault Twingo. B) Sistema estéreo y cámara a color. C) Cámara Interior. D) Cámara de infrarrojo lejano E) PC. F) Multiplexador.

Los primeros trabajos se desarrollaron sobre el primer Ivvl (Fig. 2.6. A) y estaban basados en sistemas de ayuda a la conducción mediante el análisis de imágenes y la visión por computador. En este vehículo se implementó:

- Un sistema estéreo blanco y negro (Fig. 2.6. B) que permitía, a través de un barrido progresivo, capturar imágenes en movimiento y evitar problemas inherentes al vídeo entrelazado, junto a una cámara a color (Fig. 2.6. B) para el desarrollo de tecnologías relacionadas con la señalización vertical.
- Una cámara de infrarrojo lejano (Fig. 2.6 D) que capta el calor de los objetos y una cámara en el interior del vehículo (Fig. 2.6. C), enfocando al conductor, para la monitorización del mismo y comprobar en todo momento el grado de atención del conductor.
- Todos estos elementos, junto a dos PC colocados en el maletero (Fig. 2.6. E) del vehículo para el procesamiento de los sistemas de visión por computador y un multiplexador (Fig. 2.6. F) para las señales de video que permite trabajar con ambos ordenadores de forma simultánea, son alimentados eléctricamente a través de un convertidor DC/AC conectado directamente a la batería del coche [8].

En la versión 2.0 del IvvI (Fig. 2.7. A) se trató tanto de actualizar el equipamiento como en buscar un diseño más acorde a los que en un futuro se comercializará como sistema de ayuda a la conducción. Para ello, tanto los ordenadores como los distintos sensores y monitores están perfectamente integrados dentro del vehículo [9].

A B C D

Fig. 2.7. Plataforma de Investigación IvvI 2.0 A) Vehículo Nissan Note. B) Sistema de visión estéreo. C) Pantalla. D) Ordenadores

Se dispone, como en la primera versión del vehículo, de un convertidor de potencia DC/AC conectado a una batería auxiliar que permite alimentar el conjunto de dispositivos que forman parte del vehículo:

- Dos cámaras a color. Una de ellas se encarga de monitorizar al conductor y comprobar si este se encuentra atento a la carretera y, detectar los primeros síntomas de cansancio y sueño para lanzar un aviso. La segunda detecta la señalización de tráfico y comprueba que se cumplen las normas que regulan la circulación.
- Sistema de visión estéreo (Fig. 2.7. B) para detectar obstáculos durante la conducción. Además, realiza también la odometría visual y el reconocimiento de número de carriles de la carretera con su correspondiente posicionamiento.

- Una cámara de infrarrojo encargada de detectar peatones cuando las condiciones de visibilidad son bajas o durante la conducción nocturna.
- Un sensor láser colocado en el paragolpes delantero (Fig. 2.7. A), permitiendo detectar posibles obstáculos a grandes distancias.
- Pantalla integrada (Fig. 2.7. C) en el salpicadero que permite mostrar la información captada y analizada por los sensores, además de ofrecer información en tiempo real sobre la posición de los obstáculos en el horizonte.
- Tres ordenadores (Fig. 2.7. D) situados en el maletero que analizan la información sensorial, y que pueden intercambiar información de los respectivos módulos a través de su conexión en red. En la actualidad han sido sustituidos por un solo PC de altas prestaciones, que trabaja bajo el sistema operativo Ubuntu.

El vehículo tiene instalada una sonda CAN-Bus que permite conocer la aceleración, velocidad o giro del volante entre otras muchas aplicaciones, junto con un sistema GPS-IMU del que se obtiene principalmente información de la posición.

2.3.2. iCab

También existe el Intelligent Campus Automobile o iCab (Fig. 2.8), plataforma de investigación, sobre la que se trabaja en el presente proyecto y que dada su importancia dentro del mismo será detallada más adelante, que busca desarrollar sistemas de transporte inteligentes en entornos urbanos. Se trata de un vehículo eléctrico de golf cuya función principal será llevar a los visitantes del Campus de forma autónoma de una zona a otra.

Fig. 2.8. Plataforma de investigación iCab

3. ESTADO DEL ARTE

Tanto en los sistemas avanzados de ayuda a la conducción aplicados a la mejora de la seguridad vial, como en los sistemas de navegación autónoma de vehículos, son necesarios sensores y algoritmos complejos, capaces de captar y procesar información del entorno vial e interpretarla en tiempo real para mejorar la toma de decisiones. Los principales problemas surgen a la hora de analizar información que proviene de los entornos urbanos, en donde existe una alta diversidad de elementos con distintas características [10].

En un principio los ADAS se desarrollaron para realizar tareas de supervisión de la conducción avisando al conductor de la generación de situaciones de peligro, pero no actuaban activamente sobre elementos del vehículo. En la actualidad esta tendencia está cambiando, y es cada vez más común encontrar sistemas comerciales que actúan sobre los frenos o la dirección, lo cual provoca que, desde el punto de vista tecnológico, sean necesarios algoritmos más eficaces en aspectos de fiabilidad y tiempo de respuesta, siendo un caso extremo cuando se aplican a vehículos de navegación autónoma [10]. La navegación autónoma y la planificación de la trayectoria son partes principales del desarrollo de sistemas capaces de circular sin la supervisión de un operador humano, y aquellas que suelen encontrar dificultades a la hora de definirlas e implementarlas.

Tal y como se mencionó al inicio de este apartado, los sistemas de navegación empleados en el área de la circulación autónoma demandan, especialmente en entornos urbanos, sensores capaces de obtener una gran cantidad de información, para poder realizar un correcto análisis de los mismos. Actualmente estos sistemas de navegación encargados de la detección de obstáculos y determinación del espacio libre, pueden clasificarse en dos grandes grupos según la tecnología del dispositivo que empleen, o bien basados en tecnología de *escaneo láser tridimensional* o *láser lidar (radar)*, o bien basados en cámaras y sistemas de visión por computador monoculares o estéreos, o bien una mezcla de ambas.

En cuanto a los sistemas basados en tecnología láser o radar están provistos de sensores cuyo coste de adquisición en la actualidad es considerablemente alto y, se caracterizan por poseer una alta precisión en sus medidas, además de ser sensores invasivos [10], es decir, que emplean, a la hora de obtener datos en tres dimensiones del entorno cercano, métodos activos caracterizados por actuar, enviando un rayo láser, sobre la escena.

Por el contrario, los sistemas basados en la visión por computador, principalmente implementados a través de sistemas monoculares o sistemas estéreo, son capaces de adquirir grandes cantidades de información y no son invasivos, es decir, no altera la integridad del objeto medido trabajando de forma pasiva. Esta tecnología permite con una sola cámara de video ofrecer el mismo nivel de precisión que los escáneres láser a una fracción del coste económico [11], a cambio de aumentar el computacional.

La gran diferencia económica entre una tecnología y otra provoca que cada vez más se desarrollen algoritmos que permitan a través del uso de cámaras obtener mapas de entorno que permitan al vehículo navegar por espacio libre conociendo la posición de los obstáculos.

El objetivo principal que concierne a este proyecto es elaborar el mapeado del entorno del iCab empleando un sistema estéreo y un software generado a través del framework ROS. Nuestra plataforma de investigación se moverá normalmente en un entorno urbano caracterizado por sus numerosos y variados elementos en función del tamaño, color, textura, lo que dificulta en gran medida el procesamiento de las imágenes del sistema estéreo [10].

Los sistemas de visión estereoscópica, formados por dos cámaras iguales y paralelas, son capaces de proporcionar información 3D, a base de procesar gran cantidad de información y de simular la visión del ojo humano. Frente al empleo de la tecnología láser tiene la ventaja de que permite triangular puntos para sacar distancias y obtener resultados con un alto grado de fiabilidad realizando una inversión económica menor, aunque requiere de un tiempo de procesamiento mayor debido a las grandes cantidades de datos que son capaces de adquirir y procesar. Los datos de imágenes adquiridos son procesados a través de un conjunto de paquetes de ROS que permiten generar, tras un proceso de rectificado, el mapa de disparidad necesario para el siguiente paso.

A continuación, se desarrollará un algoritmo a través del cual se transformará el mapa de disparidad en una nube de puntos 3D del ambiente de navegación del vehículo. Con la información obtenida se generará un *Fake Laser*, encargado de ir completando cuadrículas en un *Grid Map* con los obstáculos detectados acorde a la distancia, junto con una estimación de la posición del vehículo haciendo uso de ROS.

Emplear sistemas basados en ROS permite, de forma más sencilla para el desarrollador, la fusión de datos procedentes de múltiples sensores y la sincronización de tiempos de los

diferentes dispositivos. En el caso del iCab se emplea una arquitectura basada en ROS con el propósito de comunicar los procesos entre sí para perfeccionar la información y el conocimiento y, mejorar el proceso de toma de decisiones para evitar, de forma segura la colisión durante la navegación autónoma. Por tanto, hacer uso de esta arquitectura permite:

- Facilitar en el manejo de datos provenientes de distintos sensores, como el sistema binocular y el láser, de cara a elaborar aplicaciones de navegación autónoma.
- La escalabilidad y adaptabilidad a los cambios de la tecnología a bordo del iCab, para dar cabida a nuevos sensores.
- Mayores requerimientos de las aplicaciones.

3.1. iCab

El Intelligent Campus Automobile o iCab es una plataforma de investigación que busca desarrollar sistemas de transporte inteligentes en entornos urbanos. El vehículo consiste en un carrito de golf, modelo EZ-GO (Fig. 3.1), modificado mecánica y eléctricamente para operar de manera autónoma y funcional haciendo uso de un ordenador embarcado. El desarrollo del software se lleva a cabo a través del framework o plataforma de software Robot Operating System (ROS) que destaca por su portabilidad, su capacidad para procesar datos obtenidos de fusión de múltiples sensores y por la comunicación entre procesos.

Fig. 3.1. Vehículo eléctrico modelo EZ-GO

La función final de este automóvil inteligente será llevar a los visitantes del campus de una zona a otra, para lo que se dispondrá de una flota de vehículos que formarán parte de una red de transportes, siendo capaces de interaccionar de manera autónoma entre sí en ambientes próximos, facilitando la movilidad urbana.

El iCab es una de las principales vías de investigación del LSI de la Universidad Carlos III de Madrid, con el que se pretende implementar y mejorar la navegación y planificación de ruta en sistemas autónomos basados en el procesamiento de imágenes, en la visión por computador y el escaneo láser.

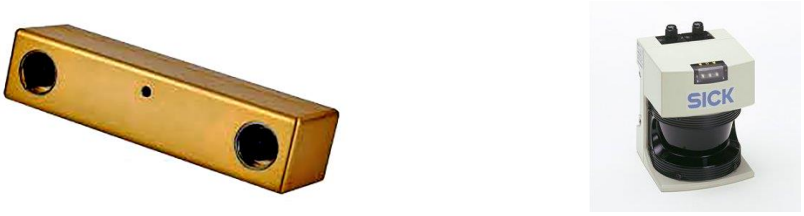
La plataforma posee una serie de modificaciones mecánicas y eléctricas para cumplir con los objetivos del proyecto en cuanto a navegación autónoma y planificación de la trayectoria se refiere. Por ello, se sustituye el volante por un sistema motor-encoder que establece un control electrónico de la dirección del vehículo (Fig. 3.2) y, se introduce un amplificador de potencia gobernado a través de un microcontrolador PIC para controlar el motor eléctrico de tracción en lugar de emplear el acelerador.

Fig. 3.2. Control electrónico de la dirección del iCab

Para garantizar una circulación autónoma segura es necesaria la percepción del medio ambiente. Para ello, el vehículo tiene instalado:

- Un telémetro láser (SICK LMS 291) (Fig. 3.3. B) en el parachoques delantero, a 30 cm de altura sobre el suelo, que cuenta con más de 180 grados de escaneo con 0.25 grados de resolución angular y con una limitación de 100 grados a 20 Hz en el rango de exploración, con el objetivo de evitar la detección de las ruedas de dirección [12].

- Una cámara binocular de visión estéreo (Bumblebee 2) (Fig. 3.3. B), colocada en el parabrisas delantero a 160 cm de altura con respecto al suelo y una orientación de -45º, con una resolución máxima de 1032x776 píxeles a 20 fotogramas por segundo [13]. A través de la información captada por este dispositivo se creará un mapa del entorno cercano al vehículo, objetivo principal del presente proyecto.



A B

Fig. 3.3. Dispositivos de la Plataforma Inteligente iCab. A) Sensor Bumblebee 2 para visión estéreo. B) Láser SICK LMS291 para detección de obstáculos

Todos los dispositivos se encuentran conectados a un ordenador de a bordo que trabaja bajo el sistema operativo Ubuntu, y además, el iCab está provisto con una pantalla táctil TFT LCD de 7 pulgadas, instalada en el salpicadero, que a través de una interfaz gráfica permite entre otras funciones, mostrar tanto la localización actual como la posición deseada en el mapa.

4. FUNDAMENTOS TEÓRICOS

En el presente trabajo se construye un mapa del entorno cercano al vehículo iCab a partir de las imágenes tomadas por una cámara estéreo. La visión estéreo continua siendo una de las áreas más estudiadas en la visión por computador, ya que permite obtener información 3D a partir de imágenes en dos dimensiones.

Para realizar el mapeado del entorno se desarrollan un conjunto de aplicaciones que permiten transformar el mapa de disparidad, generado a partir de las imágenes captadas por la cámara binocular, en una imagen en tres dimensiones, conocida como nube de puntos. Posteriormente, la información en 3D se empleará para generar un Fake Laser encargado de completar las cuadrículas en un Grid Map con los obstáculos según la distancia.

En el actual capítulo se realiza una descripción de los fundamentos teóricos utilizados a lo largo del proyecto.

4.1. SISTEMAS DE VISIÓN ESTEREOSCÓPICA

4.1.1. VISIÓN BINOCULAR

La visión binocular se caracteriza por tener órbitas oculares en el caso de seres vivos, o cámaras frontales, cuando se habla de visión en robótica o por computador. De esta manera el área de incidencia de visión de ambos ojos o sensores es prácticamente idéntica, generando una visión tridimensional del espacio visual y, siendo un tipo de visión que provoca la pérdida de amplitud de campo para ganar en profundidad.

Una gran cantidad de seres vivos, entre los que se encuentra el ser humano, perciben el mundo en tres dimensiones gracias a que poseen un sistema de visión binocular, también conocido como visión estéreo, que les proporciona un sentido de percepción de la profundidad debido a un fenómeno visual conocido como estereopsis [14]. La estereopsis es un fenómeno dentro de la percepción visual a través del cual el cerebro es capaz de recomponer una imagen tridimensional a partir de dos imágenes ligeramente diferentes del mundo físico proyectadas en la retina de cada ojo. A la diferencia entre las dos imágenes retinianas se llama disparidad binocular, y se origina por la diferente posición de ambos ojos en la cabeza [15]. Gracias a éste fenómeno, se puede calcular la posición en el mundo físico de

los objetos dada la proyección de los mismos, la cuál será ligeramente diferente en cada retina.

La visión estereoscópica es por tanto uno de los procedimientos que existen para obtener información sobre la estructura en tres dimensiones de una escena y, por tanto, de las distancias a los objetos que están presentes en la misma, lo cual resulta de gran utilidad en diversos campos de la tecnología, como la robótica, en la que un robot móvil debe disponer de información precisa del entorno que le rodea para poder realizar distintas operaciones sin riesgos.

Obtener las mencionadas distancias a los objetos es un proceso completamente natural e instantáneo en los seres vivos, pero extremadamente complejo por parte de un computador. Para lograrlo, se emplean las cámaras estéreo o binoculares (Fig. 4.1), que consisten en dos cámaras similares unidas cuyo desplazamiento mutuo es conocido.

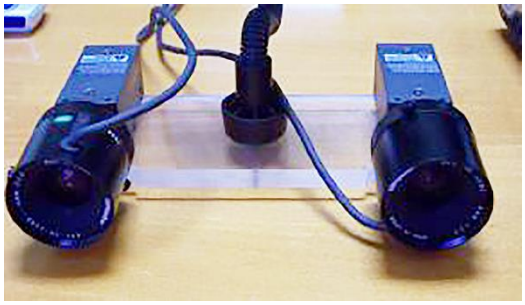


Fig. 4.1. Cámara estéreo o binocular

El campo donde confluyen cámaras y ordenadores se conoce como Visión por Computador. Las cámaras se emplean para captar las imágenes, mientras que el ordenador se requiere para realizar los cálculos que determinan la distancia al observador. Dentro de la visión por computador, los métodos de visión estereoscópica se encuentran dentro de los denominados métodos pasivos, es decir, aquellos métodos que no interfieren en la escena que están analizando.

La inmensa mayoría de los sistemas de visión estereoscópica artificial utilizan dos cámaras, tomando como referencia el modelo biológico, donde gracias a la distancia entre los dos ojos se puede establecer la distancia de los objetos, generando la tercera dimensión. Existen sistemas de visión estereoscópica artificial que en lugar de dos cámaras emplean tres, estableciendo un sistema de visión estéreo trinocular.

Los sistemas trinoculares aportan una serie de ventajas sobre los sistemas binoculares [16]:

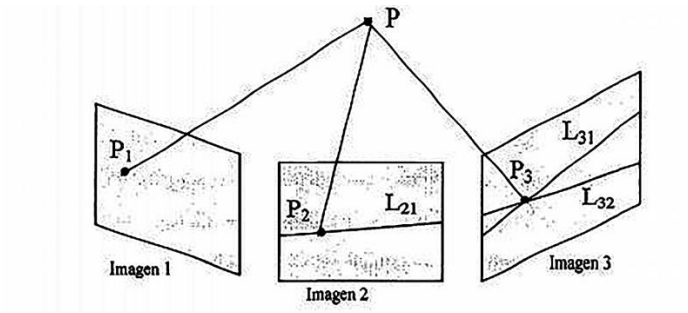


Fig. 4.2. Restricción epipolar de un sistema estéreo trinocular

- Mejora la precisión, pues los cálculos se realizan con la intersección de tres líneas de visión.
- Reduce los efectos de las oclusiones, un punto oculto en una imagen puede ser visible en las otras dos, y así calcular el punto 3D del que son proyección.
- Añade restricciones epipolares (Fig. 4.2), pues dadas dos correspondencias en dos imágenes, la tercera correspondencia en la tercera imagen se puede obtener analíticamente en el cruce de las dos líneas epipolares de las dos primeras en la tercera.

En este proyecto se centra la atención en los sistemas de visión estéreo binoculares, pues el vehículo iCab posee una cámara Bumblebee2 basada en esta tecnología.

Desde el punto de vista de la visión por computador, existen dos formas de obtener imágenes desplazadas que permitan llevar a cabo una reconstrucción en tres dimensiones:

- ❖ Visión estereoscópica basada en la toma de imágenes de dos o más cámaras alineadas y separadas una cierta distancia que debe ser conocida con exactitud. Analizando las diferencias para obtener la profundidad y centrando la atención en los efectos de la disparidad.
- ❖ Estructura del movimiento basado en una sola cámara que toma imágenes en diferentes momentos y lugares, por lo que la cámara debe ser capaz de desplazarse en línea recta y tomar imágenes mientras realiza el desplazamiento, el cual debe ser conocido. Se busca la matriz fundamental para comprender la escena.

En este proyecto, como ya se ha descrito con anterioridad, se emplea un sistema de visión estéreo formado por dos cámaras alineadas y separadas a una distancia conocida.

Los ordenadores realizan la tarea de la visión binocular buscando las correspondencias entre los puntos de una da las cámaras y esos mismos puntos en la imagen de la otra cámara. Con estas correspondencias y conociendo la distancia de separación entre las cámaras (baseline) podemos calcular la ubicación en tres dimensiones de los puntos. Para reducir el alto coste computacional que supondría buscar un punto de una imagen en la otra, se utiliza el conocimiento sobre la geometría del sistema de cámaras.

En la práctica, la visión estéreo suele englobar los siguientes cuatro pasos que reducen de manera notable el coste computacional [17]:

1. **Establecer imágenes sin distorsión.** Eliminar matemáticamente la distorsión radial y tangencial generada por la lente de la cámara, que provoca que en las imágenes las líneas rectas se muestren como curvas. Completado este proceso obtenemos un conjunto de imágenes reajustadas.
2. **Rectificación.** Consiste en ajustar los ángulos y las distancias entre las cámaras. Como resultado se obtienen imágenes rectificadas cuyas filas están alineadas.
3. **Correspondencia.** Encontrar los mismos puntos de interés en las imágenes de la cámara izquierda y derecha. Con este paso conseguimos el mapa de disparidades, siendo estas las diferencias entre coordenadas x de un punto de interés en las imágenes de la cámara izquierda y derecha: $x^{left} - x^{right}$.
4. **Reproyección.** Conociendo la disposición geométrica de las cámaras, podemos convertir el mapa de disparidad en distancias mediante la triangulación. El resultado es la posición en 3D de los puntos de interés.

4.1.2. MODELO GEOMÉTRICO DE LA CÁMARA

"Un modelo de cámara es una representación de los atributos geométricos y físicos más importantes de las cámaras utilizadas para la visión estéreo" (Montalvo Martínez, M.) [18]. En este caso se describirá un modelo que se asemeja a un sistema estéreo ideal, en el que sus dos cámaras tienen sus ejes ópticos paralelos entre sí, siendo la distancia que los separa la *baseline*, y perpendiculares a la propia *baseline*. Cualquier punto del espacio tridimensional unido a los dos centros de proyección de las cámaras define un plano epipolar. La intersección de este plano con el plano de proyección de una cámara define una línea epipolar, línea que une la imagen izquierda y la imagen derecha de un mismo punto, y que es paralela a la

baseline. Para todos los puntos, cuyas proyecciones izquierdas estén contenidas en una misma línea epipolar en la imagen izquierda, sus proyecciones derechas deben estar también contenidas sobre una misma línea epipolar en la imagen derecha, y viceversa.

El principal problema para generar la visión estereoscópica es encontrar la correspondencia entre los puntos de las imágenes [18]. Para reducir la complejidad computacional de este proceso se puede llegar a una búsqueda unidimensional, reduciéndose su vecindario a una única dimensión, para ello basta con situar y orientar las cámaras de forma que sólo exista un desplazamiento horizontal entre ellas, los ejes ópticos deben ser paralelos y los ejes de abscisas de cada una de las cámaras deben de ser coincidentes como sucede en la Figura 4.3.

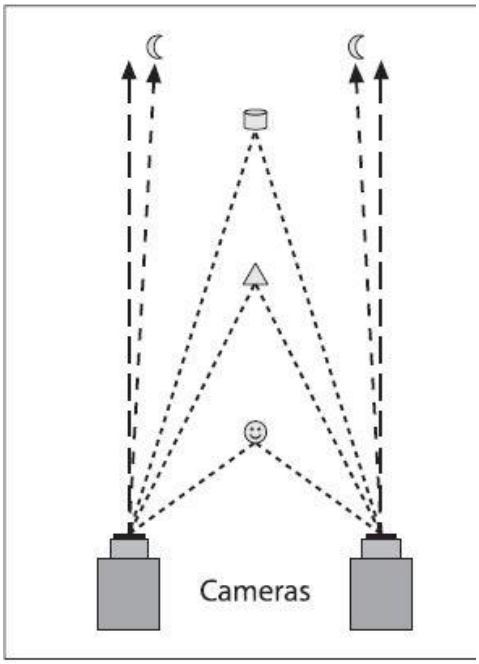


Fig. 4.3. Modelo de rayos principales paralelos [17]

Con esta distribución y orientación de las cámaras se dice que las imágenes están alineadas horizontalmente. Bajo esta situación, las líneas epipolares que definen el plano epipolar son coincidentes, facilitando la búsqueda del emparejamiento a recorrer las imágenes por filas. Con esta geometría se obtiene la denominada restricción epipolar, que ayuda a limitar el espacio de búsqueda de correspondencias, de manera que en el sistema de ejes paralelos convencional todos los planos epipolares originan líneas horizontales al cortarse con los planos de las imágenes.

En un sistema con la geometría anterior se obtiene un valor de disparidad d = X_I - X_D, para cada par de puntos emparejados P_I (X_I, Y_I) y P_D (X_D, Y_D). Con el valor de disparidad para cada punto de la imagen se crea un mapa de disparidad, en el que cada punto de la imagen contiene su valor de disparidad, y a partir del cual se logrará construir el mapa de profundidades, objetivo final de todo sistema estereoscópico.

En la Figura 4.4 tenemos un ejemplo de este tipo de sistema. El origen del sistema de coordenadas es O, la distancia focal efectiva f e igual para ambas cámaras, y la distancia entre cámaras o *baseline*, b. En cada cámara se establece un sistema de coordenadas en tres dimensiones (X, Y, Z), por lo que P_1 (X_I, Y_I) y P_2 (X_D, Y_D) son las proyecciones en las imágenes izquierda y derecha, respectivamente, de un punto P(X, Y, Z) de la escena.

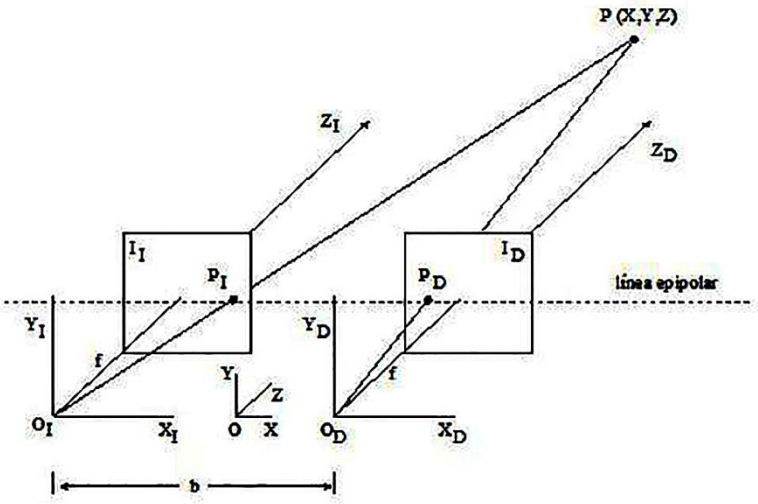


Fig. 4.4. Sistema estéreo ideal

Utilizando este sistema y la geometría del mismo, podemos obtener las siguientes ecuaciones:

$$O_I: \frac{\frac{b}{2}+X}{Z}=\frac{X_I}{f} \Rightarrow X_I=\frac{f}{Z}\left(X+\frac{b}{2}\right) \quad (1)$$

$$O_D: \frac{\frac{b}{2}+X}{Z}=\frac{X_D}{f} \Rightarrow X_D=\frac{f}{Z}\left(X+\frac{b}{2}\right) \quad (2)$$

Conociendo que la disparidad, d, es igual a $X_I - X_D$, podemos obtener el valor de la profundidad Z mediante la resta de las ecuaciones (1) y (2):

$$d = X_I - X_D = \frac{f \cdot b}{Z} \Rightarrow Z = \frac{f \cdot b}{d}$$

Este escenario se define la matriz de Reproyección Q, la cual permite que puntos en 2D puedan ser reproyectados en 3D.

$$Q = \begin{bmatrix} 1 & 0 & 0 & -c_x \\ 0 & 1 & 0 & -c_y \\ 0 & 0 & 0 & f \\ 0 & 0 & -1/T_x & (c_x - c'_x)/T_x \end{bmatrix}$$

- Donde C_x y C_y es el punto principal en la imagen izquierda
- C'_x coordenada x del punto principal en la imagen derecha. Si los rayos principales se cortan en el infinito $C_x = C'_x$
- T_x distancia horizontal entre los centros de las cámaras.
- f es la distancia focal

Con las coordenadas homogéneas en la imagen izquierda de un punto de interés y conociendo su disparidad d con su punto correspondiente en la imagen derecha, la matriz Q permite calcular coordenadas 3D en el mundo físico del punto de interés de la siguiente manera [17]:

$$Q \begin{bmatrix} x \\ y \\ d \\ 1 \end{bmatrix} = \begin{bmatrix} X \\ Y \\ Z \\ W \end{bmatrix}$$

A través de la relación establecida entre la disparidad y la distancia (Fig. 4.5) se puede deducir que para disparidades próximas a 0 la profundidad se hace muy grande ($Z \rightarrow \infty$). Esto indica que los sistemas de visión estéreo tiene una alta resolución para objetos cercanos a la cámara. Al representar como una imagen el mapa de disparidad, se puede observar que los puntos más cercanos a la cámara aparecen en un tono gris claro (valor de disparidad grande) y los puntos más alejados de la cámara aparecen en un tono de gris más oscuro (valor de disparidad más pequeño). De esta forma, los puntos que estén situados físicamente en el infinito o a una distancia suficientemente grande aparecerán en la misma posición en ambas imágenes, tendrán un valor de disparidad $d \rightarrow 0$ y en el mapa de disparidad se visualizarán en un color completamente negro.

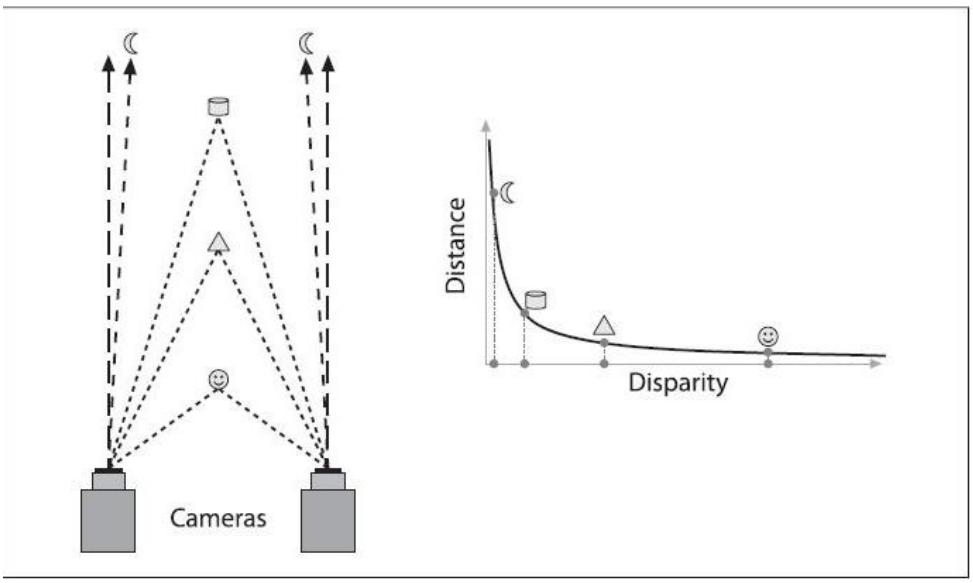


Fig. 4.5. Relación distancia – disparidad [17]

Este modelo es prácticamente ideal y en pocas ocasiones las imágenes se encuentran alineadas horizontalmente en un sistema estereoscópico.

4.1.3. GEOMETRÍA EPIPOLAR

Geometría básica de un sistema estéreo que combina dos modelos *pin-hole*, uno por cada cámara, y nuevos puntos de interés como los epipolos, que se define como la imagen del centro de la proyección de la otra cámara en un plano de proyección (π_l y π_r). Comprende una serie de reglas que permite establecer una relación entre los objetos tridimensionales y sus proyecciones en la imagen.

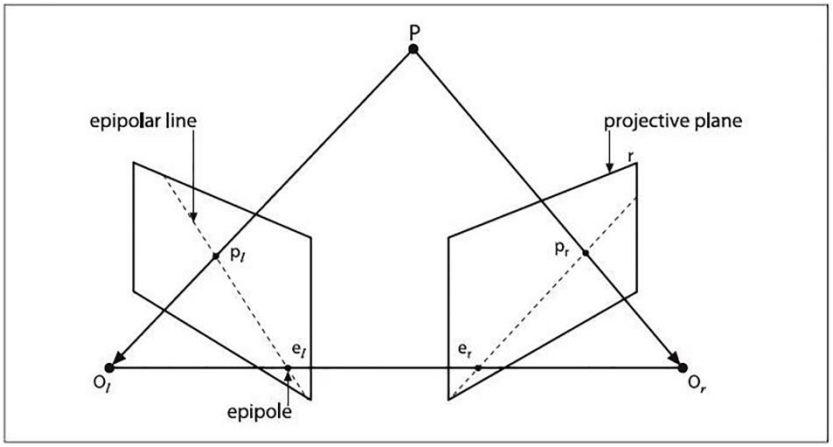


Fig. 4.6. Geometría epipolar

Como se observa en la figura 4.6 para cada cámara hay un centro separado de la proyección, O_l y O_r, y un par de planos proyectivos, π_l y π_r. El punto P en el mundo físico tiene una proyección en cada plano proyectivo, p_l y p_r. El plano formado por el punto P y los dos epipolos es el plano epipolar, y las líneas que van desde los puntos de proyección a los puntos epipolares correspondientes son las líneas epipolares.

Cuando un punto del mundo físico se proyecta sobre uno de los planos de la imagen, se desconoce la profundidad de dicho punto, ya que puede estar ubicado en cualquier parte de la recta que va de O a P_r. Esta recta contiene entre muchos otros puntos a O, y está proyectada en el otro plano siendo la línea epipolar. Todo punto del espacio tridimensional visto por las cámaras está contenido en un plano epipolar que intersecta cada imagen en una línea epipolar [17]. Esto significa que se cumple la restricción epipolar, y que la búsqueda de características en dos cámaras se reduce a una búsqueda unidimensional a lo largo de las líneas epipolares, una vez se conoce la geometría epipolar del sistema estéreo.

- ### Matriz Esencial E y Matriz Fundamental F

Dentro de la geometría epipolar se establecen la matriz esencial E y la matriz fundamental F. La matriz E posee información acerca de la translación y rotación que relaciona las dos cámaras en el espacio físico, mientras que la matriz F contiene la misma información que E más datos sobre las características intrínsecas de ambas cámaras.

- ### Calibración y Rectificación estéreo

La calibración estéreo es un proceso de cálculo de la relación geométrica entre las dos cámaras en el espacio. La rectificación estéreo tiene como objetivo hacer que las filas de las imágenes de ambas cámaras estén alineadas. El resultado de la alineación de filas horizontales dentro de un plano común de la imagen que contiene cada imagen es que los epipolos están situados en el infinito [19]. Hay un número infinito de posibles planos paralelos frontal, por lo que hay que añadir restricciones.

- **Restricción Epipolar:** Las imágenes de una misma entidad 3D deben proyectarse sobre la misma línea epipolar. Deriva, tal y como se detalló anteriormente, de la geometría de las cámaras y requiere que estas estén alineadas.
- **Restricción de Semejanza:** Ambas imágenes de una misma entidad 3D deben tener atributos similares.

- **Restricción de Unicidad:** Para cada característica en una imagen debe existir una única característica en la otra, salvo casos de oclusión donde no haya correspondencia alguna.
- **Restricción de orden posicional:** Dadas dos características en una imagen, situada una a la derecha de la otra, esta restricción supone que este mismo orden se mantiene en la imagen contraria para sus respectivas características homólogas (Fig. 4.7).

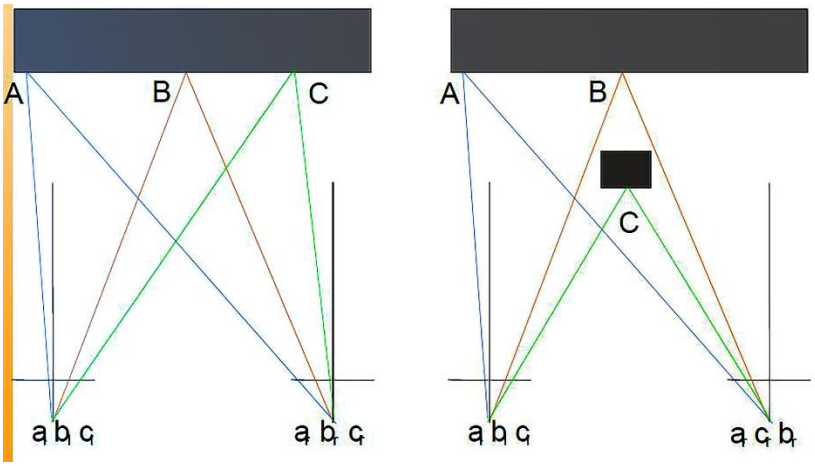


Fig. 4.7. Restricción de orden posicional

- **Restricción de continuidad de la disparidad:** asume que las variaciones de disparidad en la imagen son generalmente suaves, es decir, consideramos un mapa de disparidad continuo.
- **Relaciones estructurales:** Supone que los objetos están formados por aristas, vértices o superficies de una determinada estructura y una disposición geométrica entre los mismos.

Este apartado recoge los aspectos teóricos relativos a los sistemas de visión estereoscópicos. En él se ha detallado que los sistemas de visión por computador la información acerca de las distancias a las que se sitúan los objetos en una escena se construyen en forma de una estructura conocida técnicamente como mapa de disparidad, el cual no es ni más ni menos que una representación de las diferentes profundidades a las que se encuentran los objetos respecto a la ubicación de las cámaras. A continuación, mediante relaciones geométricas basadas en semejanza de triángulos y, conociendo los parámetros de las cámaras como puede ser la distancia de separación entre las mismas (*baseline)* o las distancias focales, es posible establecer las distancias buscadas y generar, a través de plataformas de software como ROS o PCL, una imagen en tres dimensiones conocida como nube de puntos o *point cloud.*

4.2. NUBE DE PUNTOS

La nube de puntos (Fig. 4.8) o *point cloud* consiste en una estructura de datos que representa un conjunto de puntos en varias dimensiones. Con frecuencia se emplea para reproducir datos tridimensionales [19].

Fig. 4.8. Point Cloud generada a través de ROS

En una nube de puntos en 3 dimensiones, cada punto posee tres coordenadas X, Y, Z. Si además, se dispone de la información del color, la nube de puntos pasa a convertirse en una nube de 4D. Estas nubes de puntos se pueden obtener mediante el uso de cierto tipo de sensores como son las cámaras estereográficas, las cámaras 3D o cámaras de time of flight. En este trabajo, la nube de puntos se crea mediante la transformación de un mapa de disparidad creado a partir de las imágenes captadas por una cámara binocular.

El manejo y tratamiento de las nubes de puntos se realiza a través de la Point Cloud Library (PCL). PCL es un framework independiente y de código abierto en el que se incluyen un amplio abanico de algoritmos que permiten manipular nubes de puntos en N dimensiones, y procesar de manera tridimensional las mismas.

La biblioteca PCL está desarrollada por un conjunto de ingenieros e investigadores de todo el mundo. Está escrita en C++, y al igual que ROS está distribuida bajo licencia BSD, la cual permite que sea usada libremente para propósitos, tanto comerciales como de investigación.

5. HERRAMIENTAS DE SOFTWARE

En este apartado se analizan las distintas herramientas de software empleadas para desarrollar las distintas aplicaciones que componen este proyecto. Conocer y manejar el sistema operativo Linux o la plataforma de software ROS, imprescindible para alcanzar el conjunto de objetivos que engloba el presente trabajo, fueron pasos necesarios para comenzar a implementar las primeras aplicaciones, por lo que se considera importante dedicar un apartado de esta memoria a detallar distintos conceptos de los distintos software empleados.

5.1. LINUX

El entorno empleado para desarrollar el proyecto ha sido Linux, por tratarse de un sistema operativo de gran estabilidad y compatible con todas las aplicaciones necesarias para elaborar un proyecto. En concreto se ha elegido Ubuntu en su versión 14.04 (Fig. 5.1), una distribución GNU/Linux, cuya licencia es libre y cuenta de código abierto.

Las principales ventajas de este sistema operativo, frente a otros como Windows, son:

- Es libre, multitarea, multiplataforma, multiusuario y multiprocesador.
- Protege la memoria para hacerlo más estable frente a caídas del sistema.
- Carga selectiva de programas según la necesidad.
- Uso de bibliotecas enlazadas tanto estáticamente como dinámicamente.

Fig. 5.1. Ubuntu 14.04 LTS

La elección de Ubuntu como sistema operativo, además de las ventajas descritas anteriormente, se debe en gran medida a que ROS puede ser ejecutado sobre máquinas de tipo Unix como es Ubuntu.

5.2. ROBOT OPERATING SYSTEM (ROS)

Robot Operating System (ROS) es un framework o plataforma de desarrollo de software de código abierto (OSS) específico para sistemas robóticos que se creó con el objetivo principal de apoyar el código reutilizable a la investigación robótica y el desarrollo [20]. Proporciona un conjunto de servicios y librerías reutilizables que permiten reducir considerablemente la dificultad a la hora de elaborar nuevas aplicaciones complejas para robots tanto simulados como reales.

ROS es considerado también como un meta sistema operativo de código abierto para robots, y como tal aporta los servicios que caben esperar de un sistema operativo, incluyendo abstracción de hardware, controladores de dispositivos a bajo nivel, librerías, herramientas de visualización, comunicación por mensajes, administración de paquetes mediante comandos desde terminal, compilación y ejecución de archivos bajo la licencia open source BSD.

Presenta una alta flexibilidad en cuanto al uso de distintos lenguajes de programación, soportando Python, C++, Lisp, Lua o JAVA. Puede ser ejecutado sobre máquinas tipo Unix, como Ubuntu y MAC OS X, aunque puede encontrarse soporte para otras plataformas como Gentoo o Fedora.

Consta de una estructura modular, en la que cada nodo (programa o aplicación) se ejecuta dentro de otro ejecutable (Master) que funciona de núcleo de sistema y permite, actuando como plataforma, la comunicación entre nodos mediante topics y/o servicios.

Existen diversas versiones de ROS:

- ROS Indigo Igloo (Julio, 2014)
- ROS Hydro Medusa (Septiembre, 2013)
- ROS Groovy Galapagos (Diciembre, 2012)
- ROS Fuerte (Abril, 2012)
- ROS Electric (Agosto, 2011)
- ROS Diamondback (Marzo, 2011)
- ROS C Turtle (Agosto, 2010)
- ROS Box Turtle (Marzo, 2010)

Junto al objetivo principal, mencionado al inicio del presente apartado, el marco ROS posee un conjunto de objetivos [20]:

- **Sencillez:** Diseñado para ser lo más sencillo posible de modo que el código escrito para ROS pueda ser utilizado con otras plataformas de software para robots.
- **Modelo de bibliotecas:** Los programas se escriben con interfaces funcionales y limpias que faciliten una modificación rápida de los mismos.
- **Independencia de idiomas:** ROS se puede implementar en cualquier lenguaje de programación moderno (Python, C++ o LISP, entre otros).
- **Modo test:** Posee una orden ROSTEST que permite acceder de manera sencilla a un modo de prueba de sistema.
- **Escala:** Plataforma de software adecuada para trabajar en sistemas grandes y en ejecuciones de procesos de gran tamaño.

Todas las características descritas hacen de ROS una herramienta idónea para este proyecto. Dado el peso que tiene esta plataforma en el trabajo realizado, se incluye el Anexo I al final de la presente memoria donde se detalla en profundidad la estructura y conceptos que forman parte de este software, y cuyo estudio y conocimiento ha sido necesario para poder alcanzar todos los objetivos propuestos.

5.3. Qt CREATOR

"Qt Creator es un entorno de desarrollo integrado (IDE) multiplataforma que se ajusta a las necesidades de los desarrolladores Qt" [21]. Se emplea para desarrollar interfaces gráficas de usuario y para ayudar a los programadores a utilizar programas que no poseen una interfaz gráfica, como herramientas de consola y servidores.

Utiliza de forma nativa el lenguaje de programación C++, aunque soporta otros lenguajes a través de bindings, es decir, de adaptaciones de una biblioteca para ser usada en un lenguaje de programación distinto de aquel en el que ha sido escrita.

Está disponible para los sistemas operativos GNU/Linux, Mac OS X y Windows. El API de la biblioteca cuenta con métodos para acceder a bases de datos mediante SQL, así como uso de XML, gestión de hilos, soporte de red, una API multiplataforma unificada para la manipulación

de archivos y, otros métodos para el manejo de ficheros junto con estructuras de datos tradicionales.

Está distribuida bajo los términos de GNU, siendo un software libre y de código abierto.

6. DESARROLLO Y RESULTADOS DEL PROYECTO

El presente apartado tiene como objetivo describir el conjunto de aplicaciones implementadas a lo largo del proyecto, y que permiten generar un mapeado del entorno cercano al vehículo iCab, junto con los resultados obtenidos en cada una de ellas.

El proceso que posibilita alcanzar el objetivo final del trabajo que concierne a esta memoria se compone de distintas etapas (Fig. 6.1):

- Transformar el mapa de disparidad generado a través de las imágenes captadas por un sistema de visión estéreo en una imagen en tres dimensiones o nube de puntos.

- Generar a partir de los datos obtenidos de la nube de puntos un Fake Laser.

- Emplear el Fake Laser, creado en el paso anterior, para rellenar un Grid Map con los obstáculos presentes en el entorno y en el que se muestra la localización del vehículo.

A continuación se detalla cada una de las etapas, tratando de indicar con claridad los pasos dados para alcanzar los distintos objetivos de cada una de ellas, junto con su resultado final.

Fig. 6.1. Flujograma resumen de la lógica del programa

6.1. NUBE DE PUNTOS

Para comenzar a detallar esta primera etapa del proyecto es necesario describir como se genera el mapa de disparidad, información de la cual partimos para obtener la nube de puntos, a partir de las imágenes captadas por el sistema de visión estéreo. Para la toma de datos de imagen hay tres paquetes principales (Fig. 6.2):

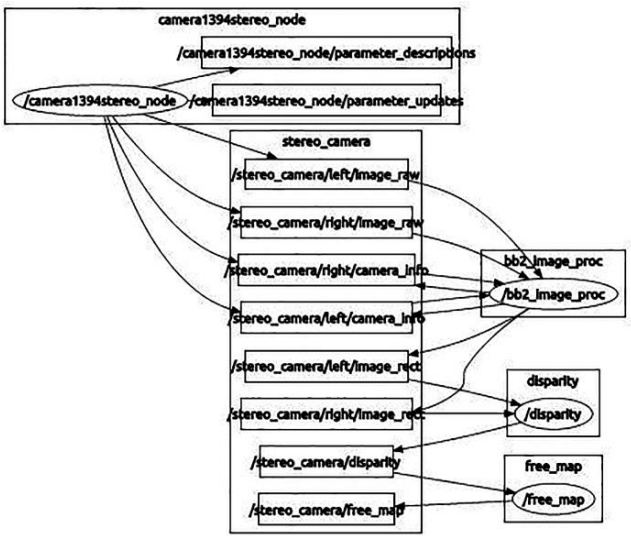


Fig 6.2. Paquetes de ROS para adquisición de imágenes

- El primero de ellos es el paquete "*camera1394stereo_node*" que recibe los datos interpolados de ambas cámaras y los divide en dos espacios diferentes, "*stereo_camera / right*" y "*stereo_camera /left*".

Fig. 6.3. Imágenes del sistema estéreo sin procesamiento previo

- El segundo paquete es el "*bumblebee2",* que recibe las imágenes izquierda y derecha del primero de los paquetes sin ningún tipo de procesamiento como "*stereo_camera / left / image_raw"* (Fig. 6.3 Izquierda) y "*stereo_camera / right / image_raw"* (Fig. 6.3 Derecha), rectifica ambas imágenes y las publica en el mismo espacio con los nombres "*stereo_camera / left / image_rect"* (Fig. 6.3 Izquierda) y "*stereo_camera / right / image_rect"* (Figura 6.4 Derecha).

Fig. 6.4. Imágenes del sistema estéreo rectificadas

Fig. 6.5. Comparación entre imagen sin procesamiento e imagen rectificada

- El tercer y último paquete es el “disparity”, que toma las imágenes rectificadas como entradas y a partir de ellas genera el mapa de disparidad (Fig. 6.6) que se emplea en esta etapa para generar una imagen en tres dimensiones.

Fig. 6.6. Mapa de disparidad

Para lograr transformar el mapa de disparidad anterior en una nube de puntos se implementará un paquete en la plataforma de software ROS. Este paquete estará formado por un único nodo, llamado P*ointcloud,* suscrito a los topic “*stereo_camera/disparity”* y “stereo_camera/left/image_rect” de los que obtendrá la información necesaria, y que publicará los resultados obtenidos durante la ejecución de la aplicación en un topic llamado *“cloud_in”,* tal y como se muestra a continuación en la Figura 6.7. Como puede verse en dicha figura es necesario que existan datos en el topic *stereo_camera / disparity* para poder generar la nube de puntos. Para ello se ejecutará de manera ininterrumpida un *rosbag*, es decir, una secuencia de video grabada de un topic de ROS, en este caso de los topics encargados de tomar datos de la cámara binocular, y que se han descrito al inicio del presente apartado. El lanzamiento del *rosbag* permite simular el proceso de captura de imágenes y creación del mapa de disparidad descrito anteriormente, para poder desarrollar la presente etapa sin necesidad de tener la cámara binocular presente y en funcionamiento, y de esta manera liberar al PC del procesamiento de esta parte.

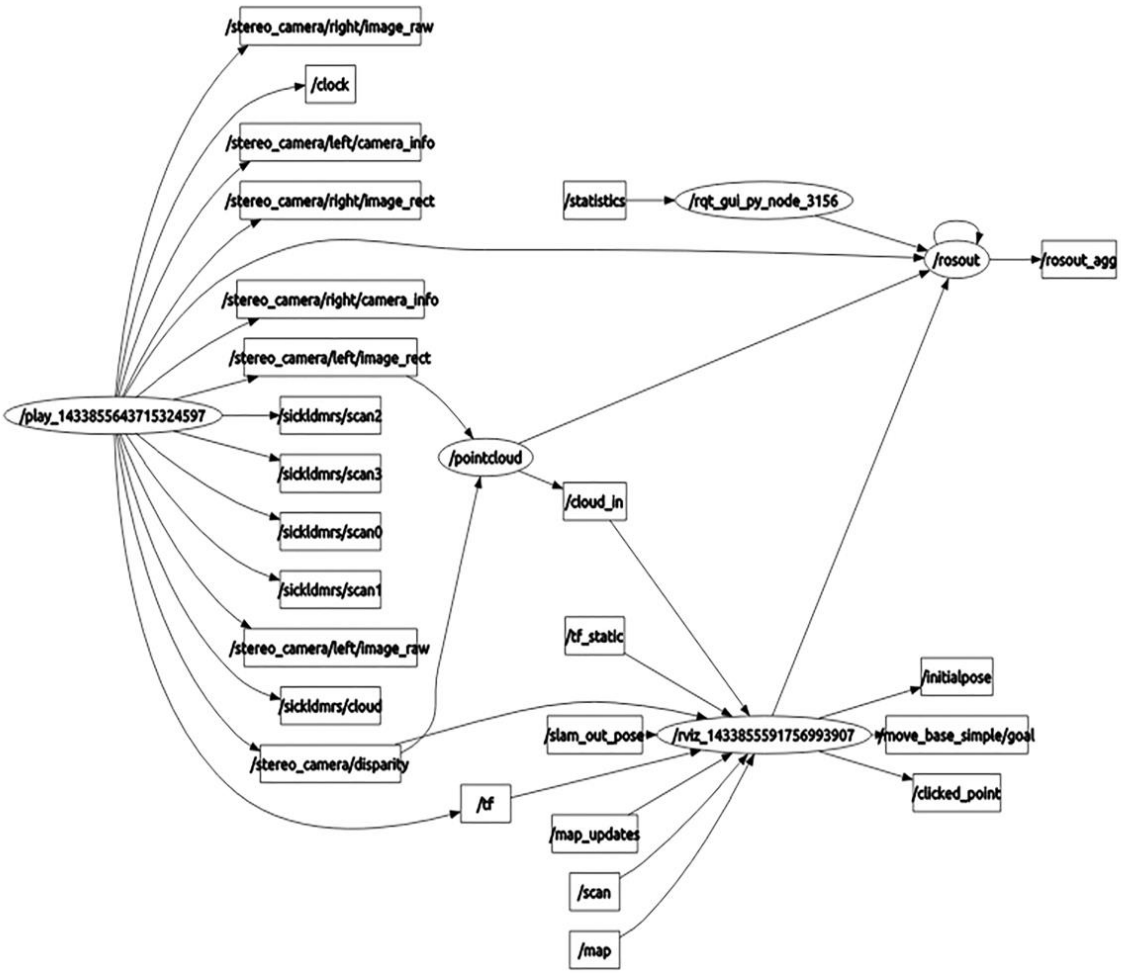


Fig. 6.7. Esquema Rqt_graph del funcionamiento de ROS durante la ejecución del paquete Pointcloud

El paquete, cuyo código se adjunta en el anexo II de la presente memoria, consta de un main y una función llamada *"pointCloudCallback"* donde se desarrollan el conjunto de instrucciones y sentencias que permiten al ejecutable o nodo generar la imagen en 3D a partir del mapa de disparidad dado.

- **Función void pointCloudCallback *(const ImageConstPtr& disp, const ImageConstPtr& l_image)***

Lo primera tarea que debe realizar esta función es convertir los datos extraídos del paquete *"disparity"* de ROS a un formato de datos compatibles con la biblioteca libre de visión artificial Open CV. Para ello haremos uso de la biblioteca *CvBridge* de ROS, la cual proporciona una interfaz entre ROS y Open CV. A continuación se generarán, haciendo uso de las librerías de Open CV, las imágenes donde se guardarán los datos extraídos de ROS y la imagen donde almacenar la nube de puntos construida durante la ejecución de la aplicación.

El siguiente paso será crear la matriz de reproyección Q necesaria para el cálculo de las distancias que permitirán generar la nube de puntos. Para construir esta matriz son necesarios una serie de parámetros de la cámara, tal y como se recoge en el punto 4.2.2 de la presente memoria, tales como:

- **Componente x del baseline (T_x):** 0.119915
- **Coordenada en el eje x del punto principal de la cámara izquierda (C_x):** 322.0614
- **Coordenada en el eje y del punto principal de la cámara izquierda (C_y):** 247.6637
- **Coordenada en el eje x del punto principal de la cámara derecha (C'_x):** 322.0614
- **Distancia focal (f):** 811.9104

El tercer paso importante de esta función será crear la nube de puntos, para lo cual se usará la función ***"reprojectImageTo3D"***, la cual recibirá como argumentos la imagen de disparidad a trasformar, la imagen en la que se guarda la nube de puntos, la matriz de reproyección Q que se ha creado anteriormente y un booleano que indicará como manejar valores perdidos, es decir, puntos en los que no se computa la disparidad. En este caso la función presentará los siguientes argumentos:

reprojectImageTo3D (disparity, XYZ, Q, true);

- **Disparity:** Imagen de entrada de disparidad.
- **XYZ:** Salida de imagen con el mismo tamaño que la de entrada. Cada elemento de XYZ contiene las coordenadas 3D del punto, calculadas a partir del mapa de disparidad.
- **Q:** Matriz de reproyección
- **True:** Indica que aquellos píxeles con disparidad mínima que se corresponden con valores atípicos se transforman en puntos 3D con un valor Z de profundidad muy grande.

Por último se establecen un conjunto de instrucciones que permiten generar la nube de puntos, y se convierten los datos de la misma al formato soportado por ROS para poder ser publicados en el topic *"Cloud_in"* y de esta manera, mediante suscripciones a este topic, poder utilizarlos posteriormente en futuras aplicaciones, como la creación de un Fake Laser.

La imagen en 3D generada durante la ejecución del nodo *pointcloud* se puede visualizar en tiempo real lanzando la herramienta Rviz (Fig. 6.8), seleccionando dentro de esta herramienta el Display "*PointCloud2"* y eligiendo el topic *cloud_in*.

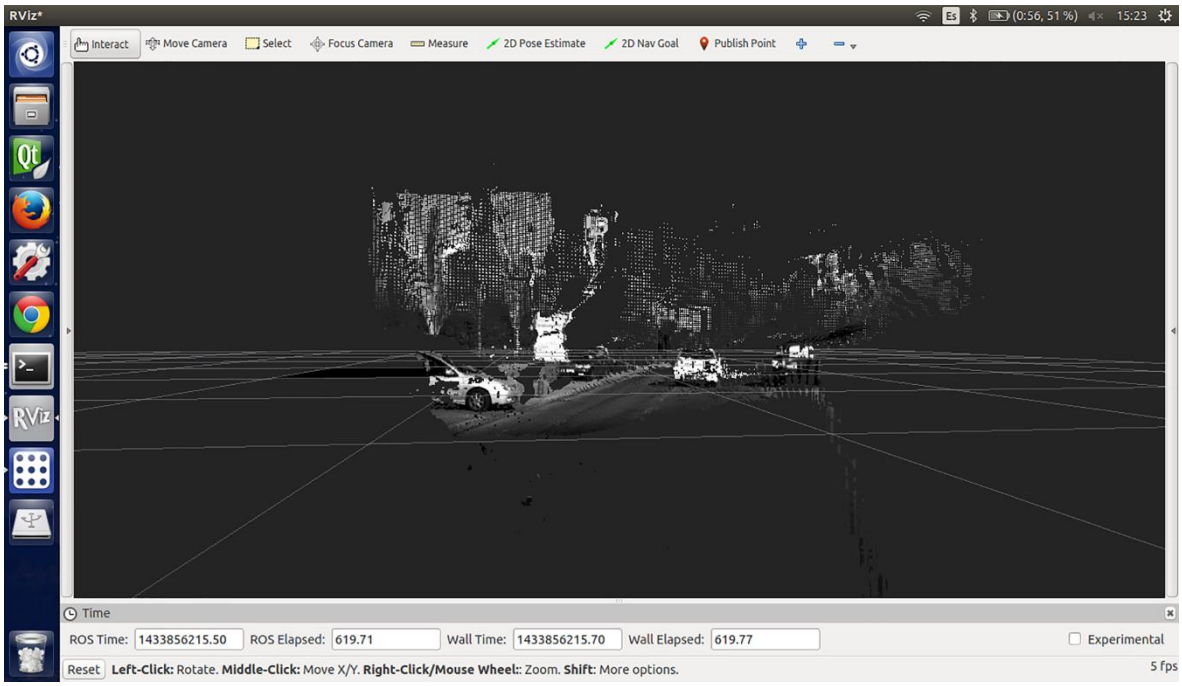


Fig. 6.8. Resultado de Nube de Puntos visualizado a través de RVIZ.

En cuanto a los resultados obtenidos, la ejecución del primero código implementado generaba una nube de puntos acorde al mapa de disparidad de entrada, pero presentaba un problema en la posición y orientación, pues las capas de la imagen 3D se creaban en una dirección distinta a la original. Para solucionarlo, se fueron modificando las relaciones establecidas en el código a la hora de introducir los datos en 3D en la estructura de la nube de puntos, hasta conseguir la orientación y posición correcta. Llevando a cabo, en las líneas de código referentes a la inserción de información en la estructura del *pointcloud,* las siguientes modificaciones:

- *point.x = px;*
- *point.y = py;*
- *point.z = pz;*

Modificaciones

- *point.x = pz;*
- *point.y = - px;*
- *point.z = - py;*

De esta manera y tal como se muestra a continuación y en la Figura 6.8, se consigue generar el conjunto de capas que forman la nube de puntos acorde al mapa de disparidad de entrada y con la misma orientación que los datos de entrada (Figura 6.9).

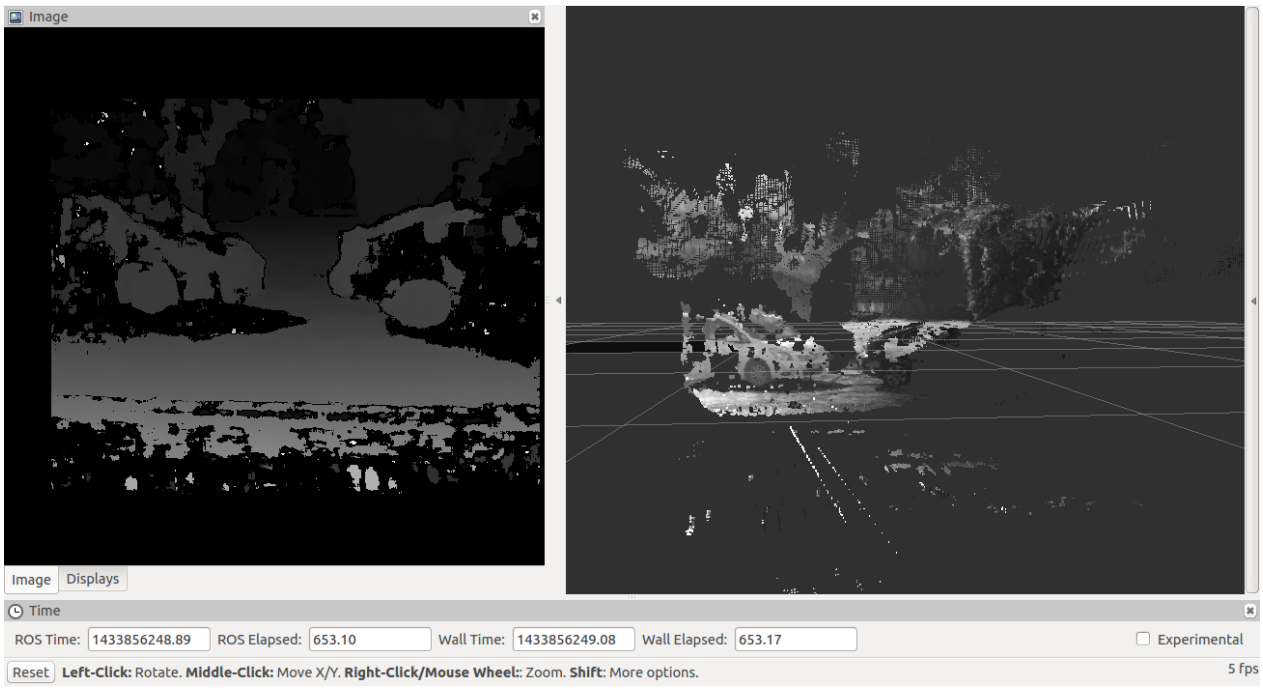


Fig. 6.9. Resultado de nube de puntos a partir de mapa de disparidad

El resultado obtenido se publica en forma de mensaje en el topic *cloud_in, y* puede visualizarse, como ya se ha mencionado, a través de la herramienta Rviz en tiempo real.

Por lo tanto, tal y como se ha descrito a lo largo del apartado, mediante la ejecución del nodo *pointcloud* se consigue cumplir el primer objetivo parcial del presente trabajo, y lograr la transformación de un mapa de disparidad en una imagen de 3D.

A continuación se detalla en un flujograma la lógica seguida a lo largo de esta primera etapa, partiendo de la captura de imágenes y yendo hasta la creación de la nube de puntos (Fig. 6.10.).

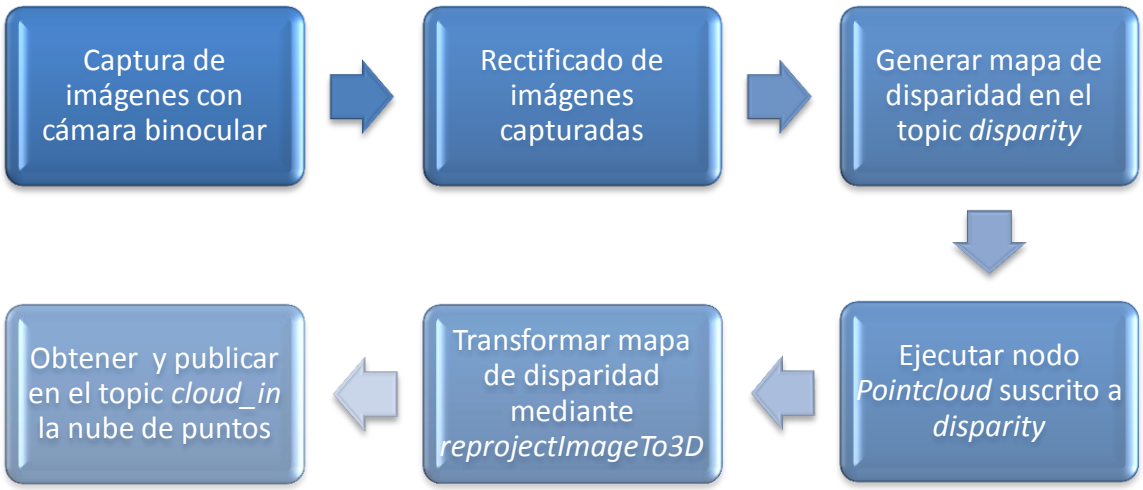


Fig. 6.10. Flujograma obtención de mapa de disparidad y transformación a nube de puntos

6.2. FAKE LASER

En este apartado se detallará la manera de obtener un Fake Laser, es decir, un escaneo láser a partir de la información obtenida de una nube de puntos.

Para realizar un escaneo láser a partir de información proveniente de la nube de puntos generada durante la ejecución del nodo *pointcloud* se hace uso del paquete *"Pointcloud_to_Laserscan"* disponible dentro de la librería de ROS. Este paquete está diseñado para convertir una nube de puntos 3D en un escaneo láser por lo que es de gran utilidad para esta etapa del proyecto, pues haciendo uso del mismo se podrá crear un *Laserscan* virtual a partir del point cloud creado en la etapa anterior.

Pointcloud_to_Laserscan tiene un nodo llamado *"pointcloud_to_laserscan_sample_node"* que al ejecutarlo permite obtener un Fake Laser de unas propiedades determinadas según el valor de los distintos parámetros que caracterizan al nodo.

El primer paso para lograr el objetivo de esta etapa será establecer una suscripción del nodo al topic *cloud_in* en el que el ejecutable pointcloud del punto anterior publica los datos de la nube de puntos, para de esta manera poder tomar los datos necesarios para generar el escaneo laser, el cual será publicado, por defecto, en el topic *scan* (Fig. 6.11).

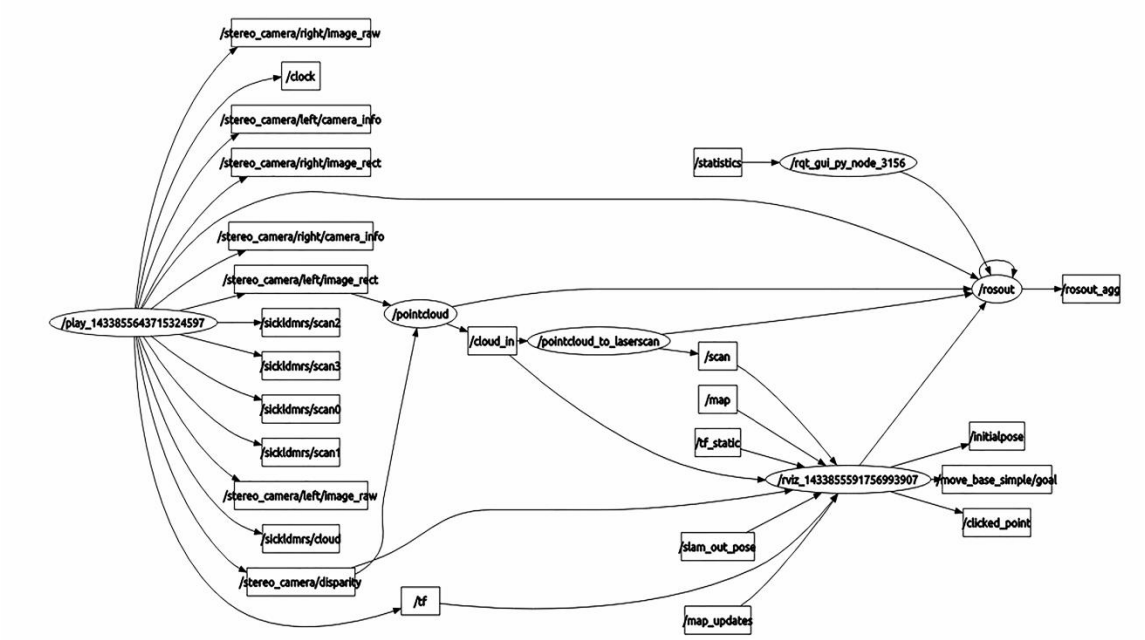


Fig. 6.11. Esquema Rqt_graph del funcionamiento de ROS durante la ejecución conjunta de los paquetes *Pointcloud y Pointcloud_to_Laserscan*

Como se observa en la figura 6.11 es necesario tener lanzado tanto el *rosbag* que simula la toma de imágenes y proceso de elaboración del mapa de disparidad, como el nodo *pointcloud* encargado de generar la nube de puntos, aunque su funcionamiento también es correcto con sensores reales.

El siguiente paso será configurar los distintos parámetros que forman parte del nodo para obtener un escaneo láser con las características adecuadas para el proyecto. A continuación se detallan los principales parámetros del nodo, indicando su función y sus valores por defecto:

- ***Min_height (0.0):*** Altura mínima de la muestra en la nube de puntos en metros.
- ***Max_height (1.0):*** Altura máxima de la muestra en la nube de puntos en metros.
- ***Angle_min (-π/2):*** Ángulo mínimo de exploración en radianes.
- ***Angle_max (π/2):*** Ángulo de escaneo máximo en radianes.
- ***Angle_increment (π/360):*** Resolución del escaneo láser en radianes por rayos.
- ***Scan_time (1.0/30.0):*** La velocidad de barrido en segundos.
- ***Range_min (0.45):*** Mínima distancia en metros para la que se realiza la detección o escaneo de obstáculos.
- ***Range_max (4.0):*** Máxima distancia en metros para la que se realiza la detección o escaneo de obstáculos.

- ***Use_inf (true):*** Parámetro empleado para gestionar zonas sin obstáculos. Si está deshabilitado la ausencia de obstáculos la reporta como ***range_max + 1,*** en caso de estar habilitado reporta un ***+inf***.

Según configuremos estos parámetros, el *Fake Laser* generado, presentará un aspecto u otro.

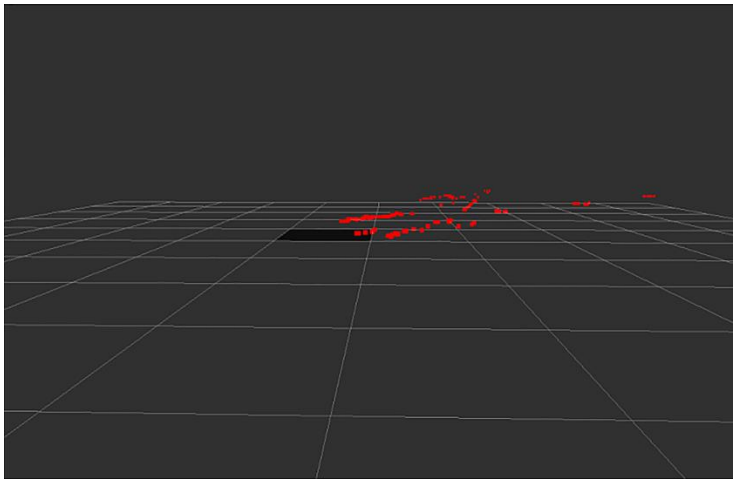


Fig. 6.12. Visualización del Fake Laser a través de Rviz

Al igual que en el apartado anterior, se podrá visualizar el resultado de la ejecución en tiempo real mediante el uso de la herramienta *Rviz,* añadiendo el Display *Laserscan,* y seleccionando el topic */scan* (Figura 6.12 y 6.13).

En este apartado se han ido obteniendo distintos resultados según se han ido modificando el valor de los distintos parámetros descritos anteriormente hasta conseguir un Laserscan virtual con la forma y características deseadas.

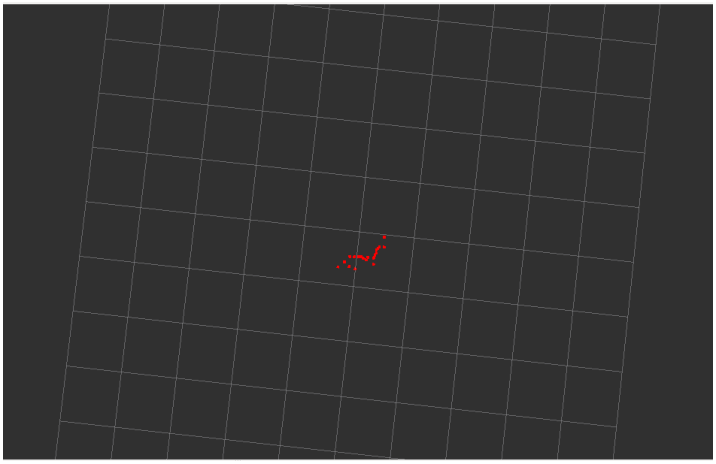


Fig. 6.13. Fake Laser (vista en planta)

Así pues, se comenzó construyendo un escaneo láser con los valores por defecto (figuran entre paréntesis en la descripción anterior de los parámetros) de los distintos factores que caracterizan al paquete. La ejecución del nodo con esta parametrización genera un Fake Laser de pequeñas dimensiones y con una baja cantidad de datos (Fig. 6.14) que resulta insuficiente para poder utilizarlo posteriormente para la creación del mapa del entorno, pero que permite ver que se trabaja con el paquete correcto y que haciendo uso del mismo se puede alcanzar el objetivo final de esta fase.

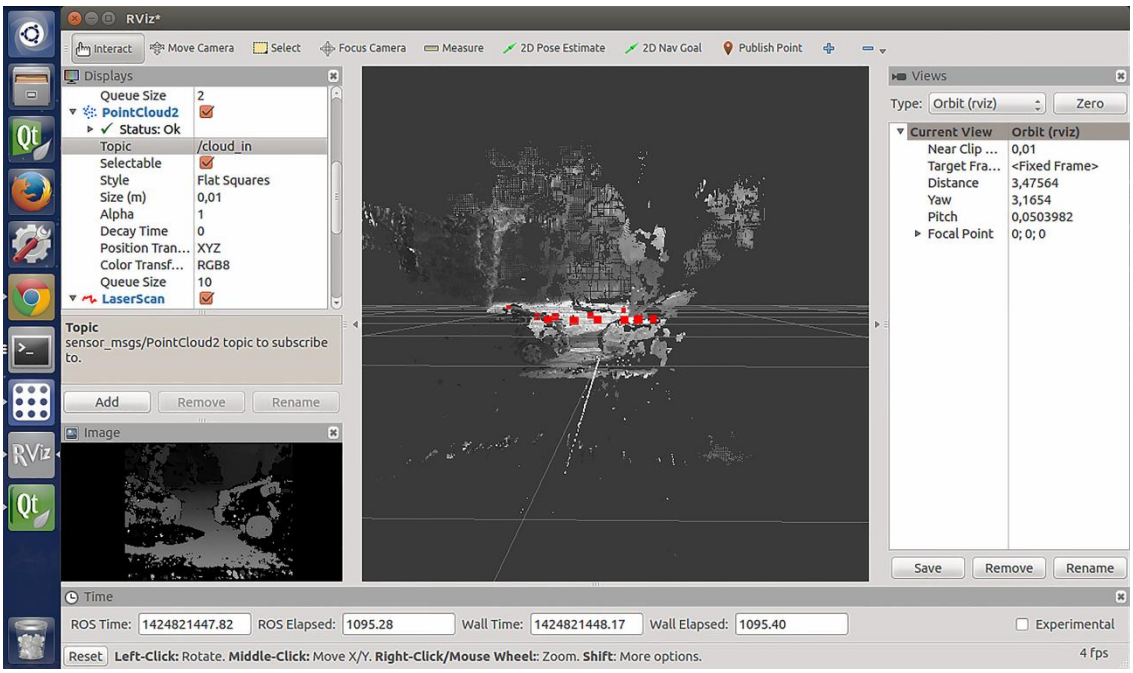


Fig. 6.14. Fake Laser con parámetros por defecto

Para obtener un Fake Laser más completo, es decir, realizar un escaneo láser con mayor cantidad de datos, modificamos aquellos parámetros que influyen directamente en este aspecto, como son el ***angle_increment*** y el ***range_max.*** Elevar el parámetro ***range_max*** provoca un aumento de la cantidad de información del Fake Laser al aumentar la distancia de detección, por el contrario es necesario reducir el valor del parámetro ***angle_increment*** para conseguir un aumento de la cantidad de datos, pues de esta manera el ángulo de incremento del escaneo es menor y se toma más información al realizar más detección en un barrido desde el ángulo mínimo hasta el ángulo máximo.

Para explicar de forma más clara el párrafo anterior se detallará el caso práctico desarrollado durante la elaboración del proyecto para cada uno de los dos parámetros mencionados con anterioridad:

- En cuanto al ***range_max,*** *se aumenta de* 4.0 metros a 10.0 metros. De esta manera se creará un escaneo láser que tenga en cuenta todos los datos de la nube de puntos presentes en el rango de distancias horizontales desde la cámara de 0.45 metros a 10.0 metros. No se tendrá en cuenta ningún dato del entorno por encima de los 10 metros de distancia.
- En el caso del ***angle_increment*** se reduce su valor de π / 360.0 a π / 540.0. Cuando se realiza un barrido se comienza tomando datos en el ángulo mínimo y se continúa el movimiento angular hacia el ángulo máximo realizando incrementos del valor del ***angle_increment,*** cada vez que se gira la cantidad de radianes que caracterizan a este parámetro se toman datos para incluirlos al Fake Laser. Así pues, cuanto menor sea el ángulo de incremento más datos se tomarán en un barrido y mayor será la información del escaneo láser (Fig. 6.15).

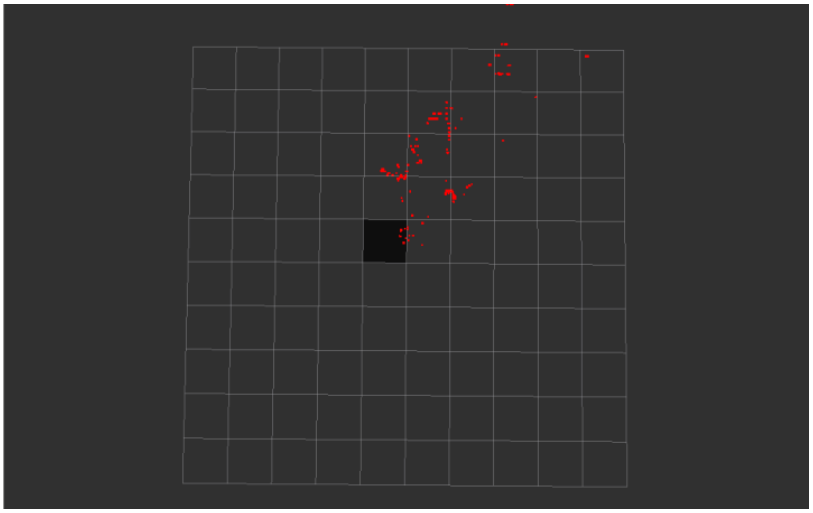


Fig. 6.15. Fake Laser actualizado (Vista en planta)

Lanzando el ejecutable del paquete *Pointcloud_to_Laserscan* con los cambios descritos, resulta un Fake Laser mucho más completo (Fig. 6.16 y 6.17), con una mayor cantidad de datos y ahora sí, con una utilidad para en la fase posterior generar el mapa del entorno.

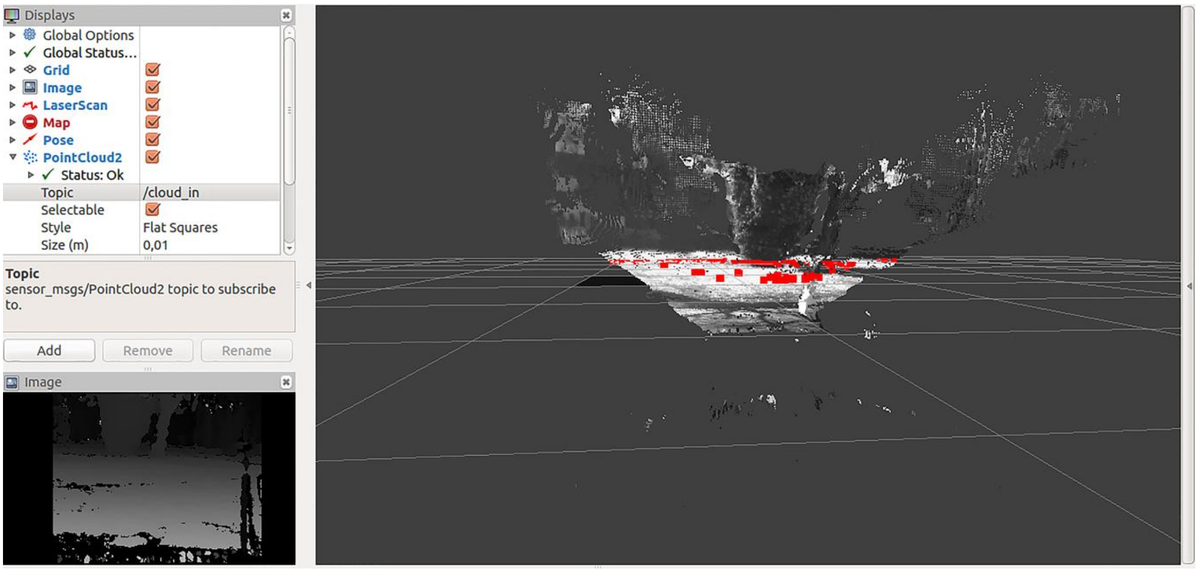


Fig. 6.16. Fake Laser (angle_increment = π / 540.0 // range_max = 10.0)

Aunque el resultado obtenido se podría considerar valido, analizando los datos del *Laserscan* (Fig. 6.17) se observa que un alto número de datos tienen el valor infinito. Esto se debe a que el parámetro ***use_inf*** posee su valor por defecto true, y por tanto, en zonas donde el escaneo no detecta nube de puntos retorna el valor **+inf**.

```
angle_increment: 0.00581000000238
time_increment: 0.0
scan_time: 0.333299994469
range_min: 0.449999988079
range_max: 10.0
ranges: [inf, inf, inf, inf, inf, inf, inf, inf, inf, inf, inf, inf, inf, inf, inf, inf, inf, inf, inf, inf, inf, inf, inf, inf, inf, inf, inf,
 inf, inf, inf, inf, inf, inf, inf, inf, inf, inf, inf, inf, inf, inf, inf, inf, inf, inf, inf, inf, inf, inf, inf, inf, inf, inf, inf, inf, in
f, inf, inf, 1.9370123147964478, 1.0684540271759033, 0.5912671089172363, 0.5909094214439392, 0.49817949533462524, 0.501456618309021, 1.00213146
2097168, 1.0084640979766846, 1.0519764423370361, 0.5488154888153076, 1.0263341665267944, 1.2285736799240112, 1.1535956859588623, 1.128708481788
6353, 1.2127894163131714, 1.2187529802322388, 1.2173370122909546, 1.2213736772537231, 1.2164636850357056, 1.1992324590682983, 1.195135831832885
7, 1.2019872665405273, 1.1985976696014404, 1.2063260078430176, 1.216977834701538, 1.225813627243042, 1.2314667701721191, 1.2150192260742188, 1.
2090563774108887, 1.2051255702972412, 1.201877236366272, 1.1973719596862793, 1.2057435512542725, 1.2025580406188965, 1.2105321884155273, 1.1937
785148620605, 1.1913084983825684, 1.186420202255249, 1.1852089166641235, 1.1926369667053223, 1.201517105102539, 1.1979396343231201, 1.210053801
53656, 1.2345170974731445, 1.24385404586792, 1.241457223892212, 1.2355525493621826, 1.1050524711608887, 4.113102436065674, 4.103615760803223, 4
.097975730895996, 4.083126544952393, 4.075807094573975, 4.221980094909668, 4.212775707244873, 4.201884746551514, 4.192938327789307, 4.185866355
895996, 4.34757661819458, 4.338654041290283, 4.154989242553711, 4.14999532699585, 3.9793405532836914, 2.1537907123565674, 3.815704107284546, 3.
811367988586426, 0.4892980456352234, 0.49547094106674194, 4.86801290512085, 4.8593831062316895, 4.429088115692139, 4.427556991577148, 4.8344140
05279541, 5.066056728363037, 5.057838439941406, 5.049785137176514, 5.041896820068359, 5.034174919128418, 5.026619911193848, 4.185139179229736,
4.009610652923584, 3.8499717712402344, 1.3331855535507202, 1.3504669666290283, 2.4322586059570312, 4.978470325469971, 4.97227668762207, 4.96625
6141662598, 4.960409641265869, 4.954737663269043, 4.950326442718506, 4.94497013092041, 4.940811634063721, 5.194511413574219, 5.189431190490723,
 1.7910221815109253, 1.7266108989715576, 1.1308282613754272, 5.459161281585693, 5.454751491546631, 3.7746009826660156, 5.159873008728027, 4.899
123191833496, 4.895840644836426, 4.892739295959473, 4.889820098876953, 4.887616157531738, 4.652884483337402, 4.8826165199279785, 4.880391120910
6445, 4.878742694854736, 2.7867698669433594, 2.7084083557128906, 2.707559585571289, 2.6337928771972656, 2.633392810821533, 2.7057266235351562,
2.6323180198669434, 2.704972267150879, 2.496642827987671, 2.562303304672241, 2.5621187686920166, 2.5621113777160645, 2.6313905715942383, 2.4965
717792510986, 2.6316981315612793, 2.6320087909698486, 2.6324188709259033, 2.6329288482666016, 2.633408546447754, 2.6340975761413574, 0.72234660
38703918, 1.4778459072113037, 1.3939404487609863, 0.9386297464370728, 1.4358116388320923, 1.4159669876098633, 1.4585380554199219, 0.62291586399
07837, 1.4387123584747314, 1.4832288026809692, 0.9243784546852112, 0.9334421157836914, 0.9436680674552917, 0.9438139796257019, 0.96292930841445
92, 1.1702344417572021, 1.4686791896820068, 0.856915295124054, 0.45861172676086426, 0.9578550457954407, 1.5434060096740723, 1.52092444896698, 1
.546342134475708, 1.5252691507339478, 1.549456238746643, 1.6016831398010254, 1.577776551246643, 1.6052013635635376, 1.7180778980255127, 0.59071
43950462341, 0.5945478677749634, 1.9255890846252441, 1.698564052581787, 0.48746374249458313, 0.48760661482810974, 0.49071288108825684, 1.503711
3428115845, 1.506508708000183, 1.6305791139602661, 7.789156913757324, inf, inf, inf, inf, inf, inf, inf, inf, inf, inf, inf, inf, inf, inf, inf
, inf, inf, inf, inf, inf, inf, inf, inf, inf, inf, inf, inf, inf, inf, inf, inf, inf, inf, inf, inf, inf, inf, inf, inf, inf, inf, inf, inf, i
nf, inf, inf, inf, inf, inf, inf, inf, inf, inf, inf, inf, inf, inf, inf, inf, inf, inf, inf, inf, inf, inf, inf, inf, inf, inf, inf, inf, inf,
 inf, inf, inf, inf, inf, inf, inf, inf, inf, inf, inf, inf, inf, inf, inf, inf, inf, inf, inf, inf, inf, inf, inf, inf, inf, inf, inf, inf, in
f, inf, inf, inf, inf, inf, inf, inf, inf, inf, inf, inf, inf, inf, inf, inf, inf, inf, inf, inf, inf, inf, inf, inf, inf, inf, inf, inf, inf,
inf, inf, inf]
intensities: []
```

Figura 6.17. Datos del topic /scan durante la ejecución del nodo *Pointcloud_to_laserscan_sample_node*

En un principio es un problema menor que podría obviarse pues no afecta a la calidad del Fake Laser obtenido, pero aumenta la carga computacional, pues se está trabajando con

información inservible. Para eliminar este contratiempo, basta con reducir los ángulos máximo y mínimo del Laserscan, haciendo que coincida el tamaño del escaneo con el de la nube de puntos, pues los datos que aparecen como ***+inf*** se deben a zonas próximas a los extremos, es decir, al ***angle_min*** y al ***angle_max*** en las que se lleva a cabo toma de datos pero en las que no existe nube de puntos.

Por consiguiente, y ya para obtener el resultado definitivo, se reduce el valor de ambos parámetros de la siguiente manera:

- ***angle_min*** *= - π / 2;*
- ***angle_max*** *= π / 2 ;*

Modificaciones

- ***angle_min*** *= - π / 3;*
- ***angle_max*** *= π / 3 ;*

Con esta nueva parametrización del nodo obtenemos, ya sí, el Fake Laser final (Fig. 6.18), que emplearemos en la próxima y última fase del proyecto para rellenar la cuadrícula de un Grid Map y generar el mapa de obstáculos.

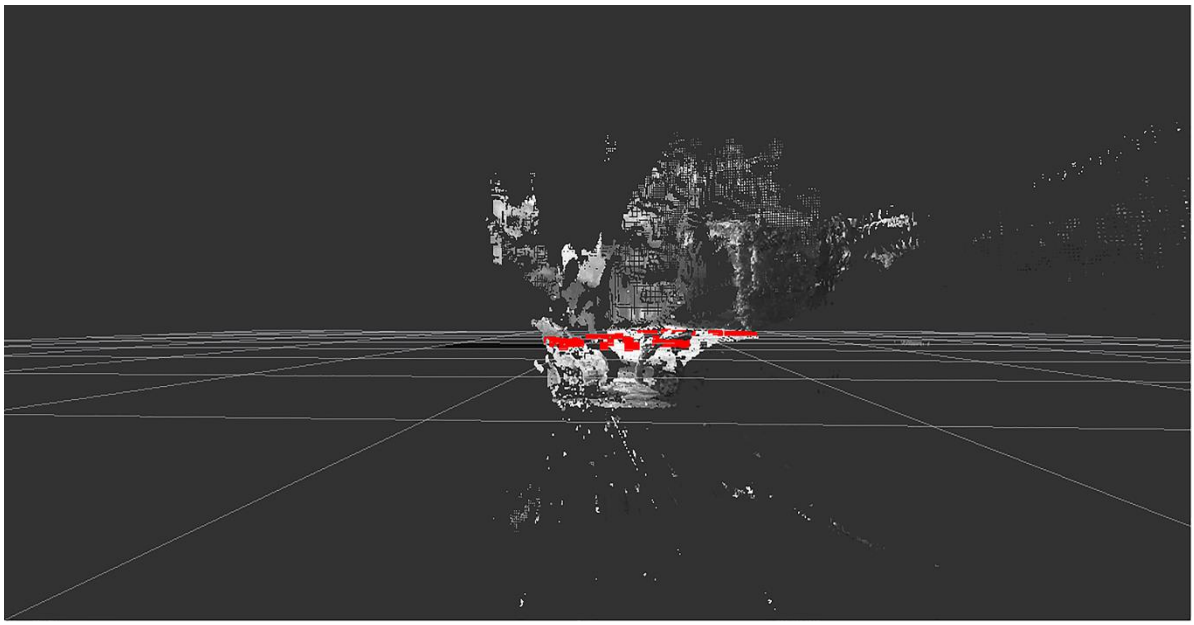


Figura 6.18. Fake Laser Definitivo

Se podrían llevar a cabo más cambios para obtener una mayor cantidad de datos en el Fake Laser, pero la cantidad de información que aportarían esos cambios no compensa al gasto computacional que supondría procesarlos.

6.3. MAPA DE OBSTÁCULOS

En este apartado se detalla la última etapa a realizar para alcanzar el objetivo final del presente trabajo, y que consiste en realizar un mapa del entorno rellenando las cuadrículas de un Grid Map con los datos obtenidos del *Fake Laser* generado en la fase anterior.

Para realizar un mapeado del entorno a partir de información suministrada por el escaneo láser generado durante la ejecución del nodo *pointcloud_to_laserscan*_node se hace uso del paquete *"Hector_mapping"* disponible dentro de la librería de ROS. Este paquete se basa en la técnica SLAM, empleada con frecuencia en robótica y vehículos autónomos, para construir un mapa de un entorno desconocido en el que se encuentra, a la vez que se realiza una estimación de la trayectoria al desplazarse dentro del entorno.

Para poder hacer uso de este paquete, es necesario un conjunto de datos que emanen de un *LaserScan*, en este caso los generados por el nodo *Pointcloud_to_laserscan_node* en el proceso descrito en el apartado anterior. Esta información se tomarán mediante la suscripción al topic */scan* donde se publica el escaneo láser obtenido en la etapa previa (Fig. 6.19).

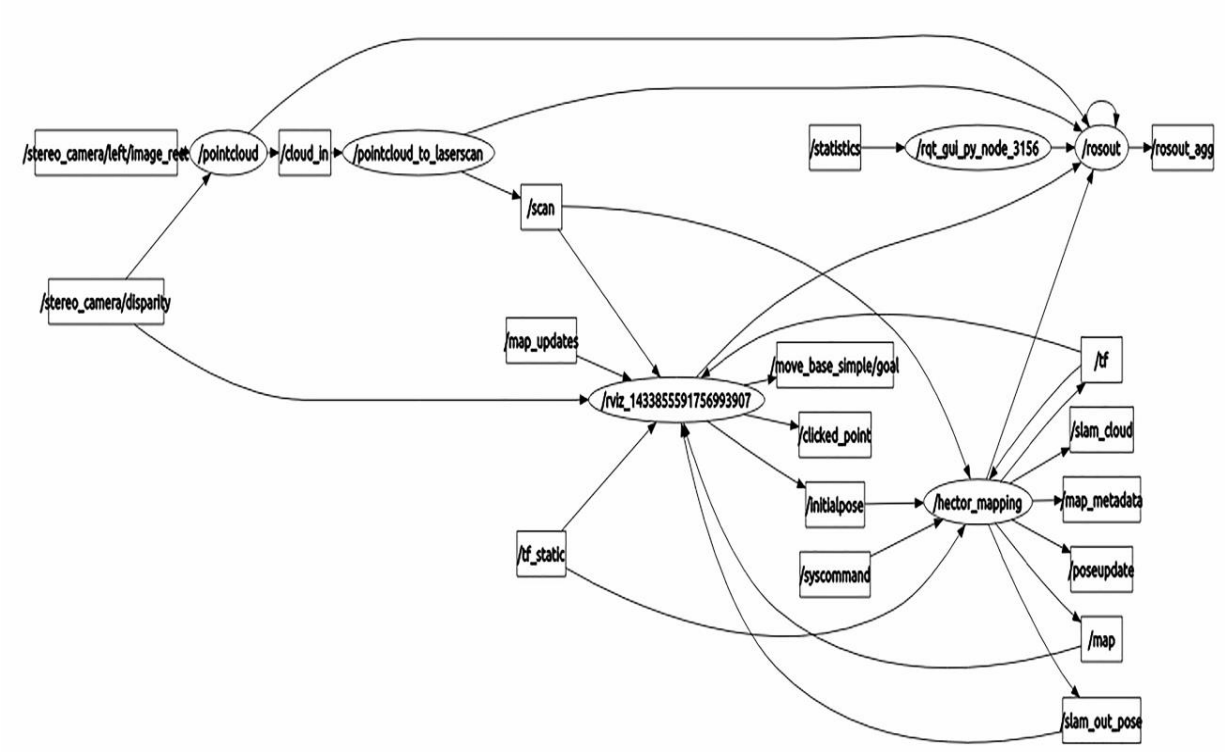


Fig. 6.19. Esquema Rqt_graph del proceso completo

Llegado este punto el coste computacional es muy alto, por lo que generar el mapa del entorno con todas las aplicaciones descritas hasta ahora en ejecución puede ocasionar problemas de velocidad en el procesamiento. Para evitar este problema, el desarrollo de este apartado se realizará empleando un nuevo *rosbag* (Fig. 6.20) que simula todos los pasos realizados hasta ahora, y ofrece un *Laserscan* generado a partir de una secuencia de video grabada con la cámara Bumblebee2 en uno de los pasillos de la Universidad Carlos III, que será el utilizado para desarrollar esta última fase del proyecto.

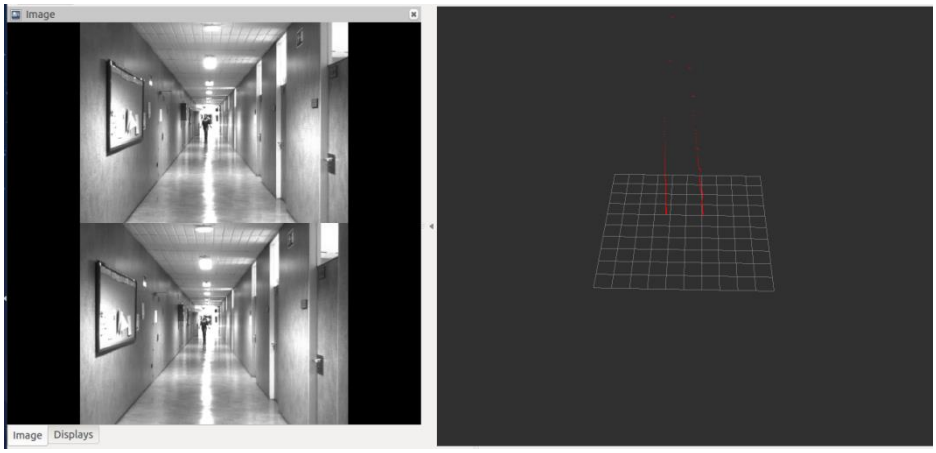


Fig. 6.20. Rosbag bagfiles_2014-09-22-12-25-31.bag

El paquete H*ector_mapping* presenta una serie de parámetros que permitirán conseguir distintos resultados según los valores que se asignen a cada uno de ellos. A continuación se detallan los principales parámetros del nodo, indicando su función y sus valores por defecto:

- ***Map_resolution (0.025):*** Resolución del mapa en metros. Corresponde con la longitud del borde de cada una de las celdas de la cuadrícula.
- ***Map_size (1024):*** Tamaño en número de células por eje, del mapa. El mapa es cuadrado y presentará una rejilla del tamaño map_size x map_size.
- ***Map_start_x (0.5):*** Localización del origen [0.0, 1.0] del mapeado del entorno en el eje X con respecto al mapa de la red, siendo 0.5 el medio.
- ***Map_start_y (0.5):*** Localización del origen [0.0, 1.0] del mapeado del entorno en el eje Y con respecto al mapa de la red, siendo 0.5 el medio.
- ***Map_update_distance_tresh (0.4):*** Umbral para realizar actualizaciones del mapa en metros. La plataforma o robot tiene que recorrer tantos metros desde la última actualización para que se actualice el mapa.

- ***Map_update_angle_tresh (0.4):*** Umbral para realizar actualizaciones del mapa en radianes. La plataforma o robot tiene que experimentar un cambio angular de tantos radianes, como el parámetro indica, desde la última actualización para que se actualice el mapa.
- ***Map_pub_period (2.0):*** Periodo de publicación del mapa.
- ***Update_factor_free (0.4):*** Modificador de mapas para las actualizaciones de celdas libres en el rango [0.0, 1.0]. Un valor de 0.5 significa que no hay cambios.
- ***Update_factor_occupied (0.9):*** Modificador de mapas para las actualizaciones de celdas ocupadas en el rango [0.0, 1.0]. Un valor de 0.5 significa que no hay cambios.
- ***Laser_min_dist (0.4):*** Distancia mínima en metros para la que se toman valores del escaneo láser. Puntos del Fake Laser a menos distancia de la indicada por este parámetro se ignoran.
- ***Laser_max_dist (30.0):*** Distancia máxima en metros para la que se toman valores del escaneo láser. Puntos del Fake Laser a mayor distancia de la indicada por este parámetro se ignoran.
- ***Laser_z_min_value (-1.0):*** La altura mínima en metros con respecto al bastidor del escáner láser para los puntos de escaneo del láser utilizados por el sistema. Criterios de valoración de escaneo inferior a este valor se ignoran.
- ***Laser_z_max_value (1,0):*** La altura mínima en metros con respecto al bastidor del escáner láser para los puntos de escaneo del láser utilizados por el sistema. Criterios de valoración de escaneo inferior a este valor se ignoran.
- ***Pub_map_odom_transform (true):*** Determina si deben publicarse por parte del sistema las transformaciones de mapa → odom.

Si se ejecuta el nodo del paquete con estos parámetros aparece un aviso en relación al último parámetro y los problemas derivados de transformaciones entre distintos sistemas de referencia. Este aviso no detiene la ejecución y permite obtener un mapa del entorno (Fig. 6.21), aunque el procesamiento de estos avisos ralentiza la visualización del mapa, por lo que se modifica este parámetro a su otro estado posible, es decir, false.

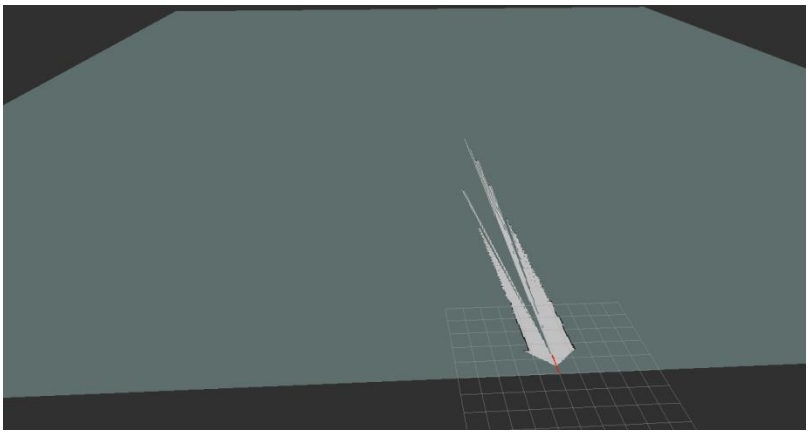


Fig. 6.21. Mapa del entorno del pasillo.

Como se puede observar en la Figura 6.21, es posible visualizar en tiempo real la generación del mapa del entorno, para lo cual se tendrá que seleccionar el Display *map* y elegir dentro del mismo el topic */map*. También se puede conocer una estimación de la posición del vehículo, para lo cual debemos introducir en la herramienta Rviz el Display *Pose* y dentro de ese Display seleccionar el topic */poseupdate.*

Para obtener un mejor mapa del entorno se aumenta el tamaño y la resolución del mapa a 2048 y 0.05 m, y estableciendo el origen del mismo en el centro del eje X y al inicio del eje Y (0.5, 0.0). Además, para generar un mapa más fiable, y tratando por tanto de procesar un mayor número de datos, se establece un valor de **40.0 m** para el parámetro ***laser_max_dist,*** un valor de **0.1** para el parámetro ***map_update_distance_tresh*** y un valor de **0.06** para ***map_update_angle_tresh.*** Esta parametrización del paquete ofrece un resultado cuyo grado de fiabilidad es bastante alto, pues posee un alto nivel de semejanza con el Fake Laser y con el entorno real (Fig. 6.22).

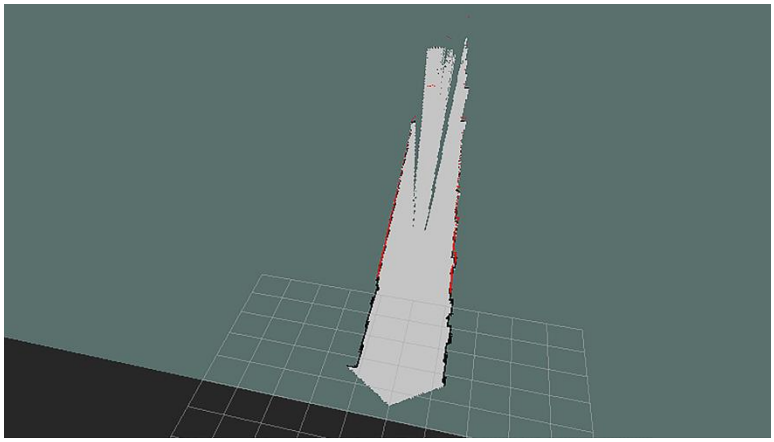


Fig. 6.22. Mapeado del entorno junto con Fake Laser de entrada.

Con las características descritas en el párrafo anterior se obtiene como resultado el mapa mostrado en la Figura 6.22 y en la secuencia de imágenes presentes en la Figura 6.23 en la que se puede ver cómo se va produciendo tanto la actualización del mapa como de la posición del vehículo.

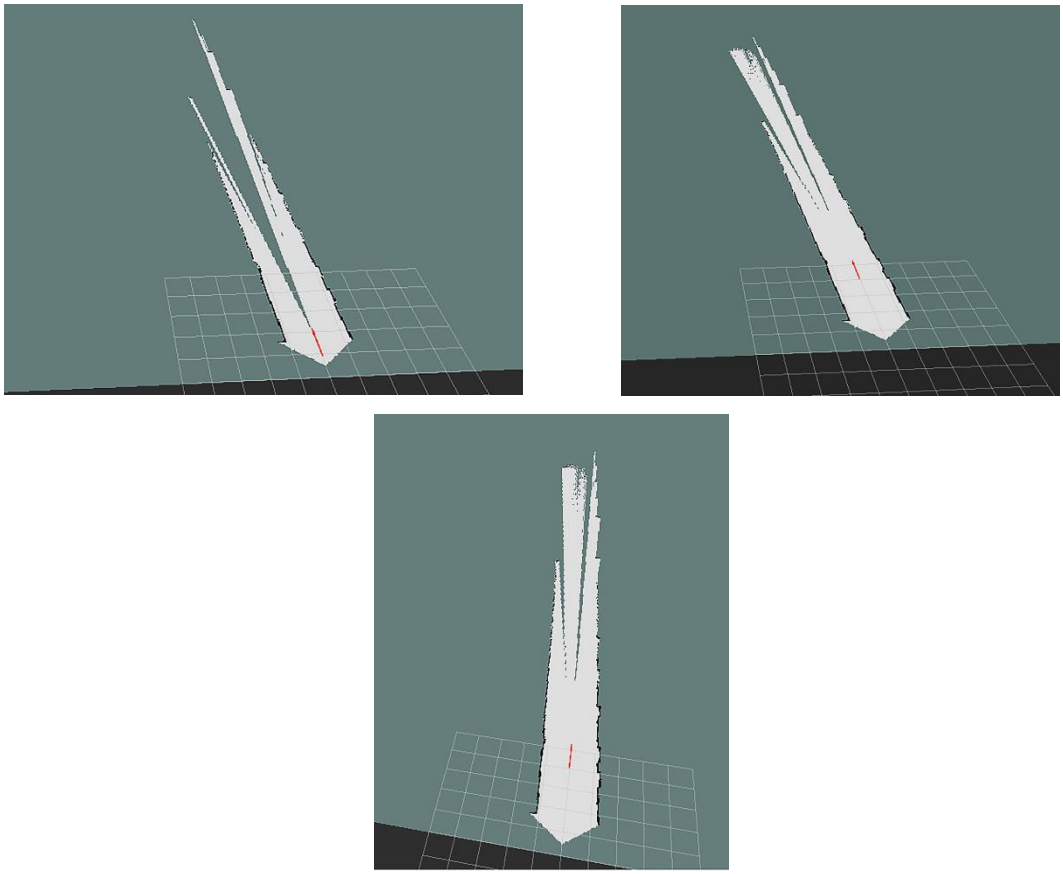


Fig. 6.23. Secuencia de imágenes de la creación del mapa del entorno del vehículo

El mapa resultante (Figura 6.23) de la ejecución del nodo parametrizado tal y como se ha descrito anteriormente, y como se recoge en el anexo de la presente memoria, es una solución óptima al problema que concierne a este proyecto, ya que se genera un mapeado del entorno cercano al vehículo en el que se puede observar claramente el espacio libre (área gris) y los obstáculos (puntos negros) que se encuentran en el horizonte de la cámara.

Por último, se sitúa el origen del mapa en el centro de la cuadrilla, para lo cual se modifica el parámetro ***Map_start_y*** *de 0.0 a 0.5,* y probamos el paquete y la configuración que se ha realizado en el mismo en conjunto con todos los procesos y aplicaciones desarrollados a lo largo del proyecto, utilizando en este caso el *rosbag* que contiene la secuencia de video de los alrededores del campus, y que solo simula el proceso de creación del mapa de disparidad.

En la Figura 6.24 se puede observar un conjunto de imágenes capturadas durante la creación del mapa del entorno haciendo uso de todas las aplicaciones desarrolladas a lo largo del trabajo.

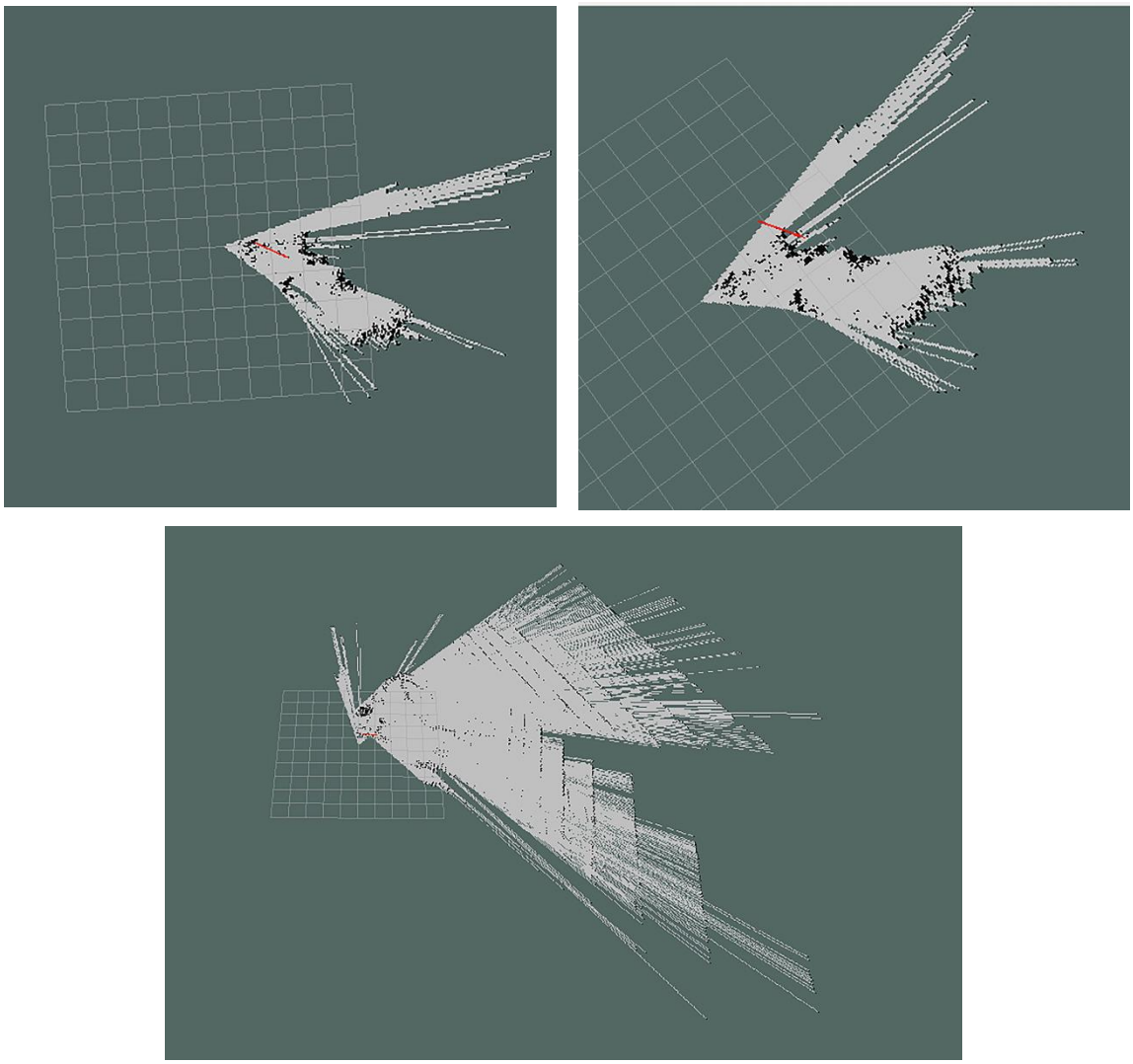


Fig. 6.24. Secuencia de imágenes de la visualización del resultado final del mapeado del entorno

7. CONCLUSIONES Y FUTUROS TRABAJOS

7.1. CONCLUSIONES

El objetivo principal de este proyecto era el mapeado del entorno cercano a la plataforma inteligente iCab a partir de la información tomada por una cámara binocular. Este requisito se ha cumplido haciendo uso del paquete *Hector_mapping,* perteneciente a la librería de ROS, que permite generar el mapa de obstáculos presentes en el horizonte de la cámara junto con una estimación de la posición del vehículo.

Además se ha alcanzado el objetivo principal resolviendo diferentes tareas previas necesarias para construir el mapa del entorno como son:

- Transformar un mapa de disparidad en una imagen 3D o nube de puntos.
- Generar un Fake Laser o escaneo láser virtual a partir de la información contenida en la nube de puntos generada en el paso anterior.

Todos estos problemas han sido resueltos haciendo uso de la plataforma de software ROS, cuyo aprendizaje, estudio y manejo también han formado parte de los conocimientos y facetas adquiridas durante el desarrollo del presente proyecto.

En resumen, se ha logrado cumplir con los principales objetivos del proyecto, empleando una herramienta especializada en el desarrollo de aplicaciones para la robótica y con la cual no había trabajado anteriormente.

7.2. TRABAJOS FUTUROS

El mapa de entorno obtenido puede emplearse, llevando a cabo una distinción entre obstáculos móviles y estáticos, para elaborar trayectorias que garantice un movimiento seguro del vehículo entre distintos puntos del campus. Además, los resultados finales obtenidos pueden fusionarse con datos extraídos del telémetro láser para generar un sistema de navegación autónoma seguro y fiable.

8. COSTES DEL PROYECTO

El objetivo de este capítulo es establecer una relación aproximada de los costes del proyecto. En primer lugar, se detallan los tiempos dedicados a cada fase del proyecto:

ETAPA	TIEMPO
Estudio, comprensión y planteamiento del problema.	10 horas
Iniciación al manejo de Linux y ROS (Tutoriales)	25 horas
Documentación	30 horas
Implementación del código	10 horas
Comprensión del código presente en los paquetes de ROS empleados	20 horas
Pruebas del algoritmo	20 horas
Mejoras para optimización de las aplicaciones (ajuste de parámetros, modificación de código)	25 horas
Resultados	10 horas
Memoria del proyecto (parte teórica)	30 horas
Memoria del proyecto (parte práctica y Anexos)	30 horas
TOTAL	210 HORAS

Suponiendo un coste aproximado de 15 € / hora para un ingeniero, el precio total de la mano de obra para el presente proyecto sería:

$$210\ horas \cdot 15 \frac{€}{hora} = 3150\ €$$

Además se han empleado los siguientes materiales:

CONCEPTO	COSTE
PC estándar	1.000 €
Cámara estéreo (Bumblebee2 de Pointgrey)	3.000 €
Paquetes de librerías ROS	Gratuito
Paquetes de librerías Open Cv	Gratuitos
Otros materiales (Cables, soportes, etc.)	100 €
TOTAL	4.100 €

Estimando que en realizar la instalación del equipo en la plataforma de investigación iCab se tardan 10 horas por parte de un técnico especializado, que cobra a razón de 30€ / hora, se tiene un coste de:

$$10\ horas \cdot\ 30 \frac{€}{hora} = 300\ €$$

Por tanto, el coste total del proyecto asciende a:

$$\boldsymbol{TOTAL = 3.150€ + 4.100\ € + 300\ € = 7.550\ €}$$

9. BIBLIOGRAFÍA

[1] Oficina Europea de Estadística, EUROSTAT 2009, Panorama de los transportes, 2009th edn.

[2] Organización Mundial de la Salud 2014, Las 10 causas principales de defunción en el mundo (200-2012), Nota Descriptiva nº310.

[3] Stanley, M. 2014, TSLA's New Path of Disruption.

[4] Milanés, V., Naranjo, J.E., González, C., Alonso, J., García, R. & Pedro, T. 2008, Sistemas de Posicionamiento para Vehículos Autónomos, Revista Iberoamericana de Automática e Informática Industrial.

[5] López Montes, D. 2014, Sistema de Control Longitudinal para Vehículo Eléctrico Urbano, Universidad de Cantabria.

[6] García Fernández, F. 2015, Vehículos Autónomos. Del Grand Challenge al Google Car, Universidad Carlos III de Madrid.

[7] López Arredondo, J. 2015, El coche autónomo de google ya tiene fecha de lanzamiento, España.

[8] Laboratorio de Sistemas Inteligentes, U.Carlos III de Madrid 2013, 06/05/2013-last update, Sistemas Inteligentes de Transporte. IvvI. Available: http://portal.uc3m.es/portal/page/portal/dpto_ing_sistemas_automatica/investigacion/lab_sist_inteligentes/sis_int_transporte/vehiculos/ivvi/ [2015, 01/06/2015].

[9] Laboratorio de Sistemas Inteligentes, U.Carlos III de Madrid 2013, 06/05/2013-last update, Sistemas Inteligentes de Transporte. IvvI 2.0. Available: http://portal.uc3m.es/portal/page/portal/dpto_ing_sistemas_automatica/investigacion/lab_sist_inteligentes/sis_int_transporte/vehiculos/IvvI20/ [2015, 01/06/2015].

[10] Musleh, B., De la Escalera, A. & Armingol, J.M. 2012, Detección de obstáculos y espacios transitables en entornos urbanos para sistemas de ayuda a la conducción basados en algoritmos de visión estéreo implementados en GPU, 9th edn, SciVerse ScienceDirect, Revista Iberoamericana de Automática e Informática Industrial.

[11] Bravo, R. 2015, 05/02/2015-last update, Desarrollo de sistemas de navegación de bajo costo para vehículos autónomos. Available: http://www.tyniot.com/desarrollan-sistema-de-navegacion-de-bajo-costo-para-vehiculos-autonomos/ [2015, 05/06/2015].

[12] Intelligence, S.S. 2006, LMS200/211/221/291 Laser Measurement Systems, Technical Description edn, Alemania.

[13] Grey, P. 2012, Bumblebee: stereo vision camera system, Technical Description edn.

[14] Peris Martorell, M. 2011, Aproximación del Mapa de Disparidad Estéreo Mediante Técnicas de Aprendizaje Automático, UPV.

[15] Tortajada Montañana, I. 2006, Expressió gráfica i infografía, 1st edn, Universidad Politécnica de Valencia, Valencia.

[16] Moreno Díaz, A.B. 2004, Reconocimiento Facial Automático mediante Técnicas de Visión Tridimensional, Universidad Politécnica de Madrid.

[17] Bradski, G. & Kaehler, A. 2008, Learning Open CV: Computer Vision with the OpenCV Library. 1st edn, O'Reilly Media.

[18] Montalvo Martínez, M. 2010, Técnicas de visión estereoscópica para determinar la estructura tridimensional de la escena., Universidad Complutense de Madrid.

[19] Nuño Simón, J. 2012, Reconocimiento de objetos mediante sensor 3D Kinect, Universidad Carlos III de Madrid.

[20] García Cazorla, A. 2013, ROS: Robot Operating System, Universidad Politécnica de Cartagena.

[21] Wiki Qt Creator2015, 24/05/2015, last update. Available: https://wiki.qt.io/Category:Tools::QtCreator_Spanish [215, 06/09].

[22] Llamazares, A., Molinos, E., Ocana, M. & Herranz, F. 2014, Integrating absynthe autonomous navigation system into ros, Internacional Conference on Robotics and Automation (ICRA 2014), IEEE.

[23] Zaman, S., Slany, W. & Steinbauer, G. 2011, "ROS-based mapping, localization and autonomous navigation using a pioneer 3-dx robot and their relevant issues." Saudi International Electronics, Comunications and Photonics Conference. IEEE, , pp. 1-1-5.

ANEXO I – ROBOT OPERATING SYSTEM (ROS)

I.I. FUNCIONAMIENTO DEL SISTEMA

ROS es una plataforma de software estructura en tres niveles de conceptos [20]: el nivel de sistema de archivos, el nivel de computación gráfica y el nivel comunitario.

- **Sistema de Archivos**

Referido principalmente a recursos que se encuentran en el propio programa:

- **Paquetes:** Unidad principal para organizar software en ROS. Puede contener procesos ejecutables o nodos, una biblioteca dependiente, un conjunto de datos, archivos de configuración, o cualquier elemento útil para la organización conjunta.
- **Manifiestos:** Proporcionan metadatos sobre un paquete, incluyendo nombre, versión, descripción, información de licencia y dependencias e información específica del compilador.
- **Pilas:** Agrupación de paquetes con una misma función.
- **Manifiestos de pilas:** Proveen datos sobre una pila, entre los que se encuentra información de licencia y sus dependencias en otras pilas.
- **Mensajes:** Definen las estructuras de los datos para mensajes enviados en ROS.
- **Servicios:** Establecen la solicitud y estructuras de los datos de respuesta de los servicios requeridos por ROS.

- **Computación a nivel gráfico**

Nivel encargado de procesar todos los datos. Los conceptos básicos que proporcionan los datos de diferentes maneras son:

- **Nodos:** Procesos que realizan cálculos.
- **Maestro:** El maestro proporciona registro de nombres y la búsqueda para el resto de la computación gráfica.
- **Mensajes:** La comunicación entre nodos se establece a modo de mensaje, el cuál es una estructura de datos que comprende los tipos de campo.
- **Temas:** Bus empleado por los nodos para enviar o recibir los mensajes.

- **Nivel comunitario**

Son recursos que permiten el intercambio de software y conocimientos:

- **Distribución:** Colecciones versionadas en pilas que se pueden incorporar haciendo más fácil la instalación de un conjunto de programas.
- **Repositorios:** Red de repositorios de código, donde las instituciones pueden distribuir sus componentes de software.
- **Wiki de ROS:** Foro principal para documentar información sobre ROS. Junto al blog de Willow Garage, creador de ROS, constituye la principal fuente de información de la plataforma.

I.II. CONCEPTOS BÁSICOS

En el presente apartado se detallará en profundidad recursos mencionados en el apartado anterior, y que forman parte de ROS y de sus niveles de concepto.

a. Sistema de archivos

El sistema de archivos de ROS se divide en dos niveles:

- **Packages (Paquetes):** Corresponde al nivel más inferior de organización del sistema de archivos. Puede contener nodos, modelos, tipos de mensajes y servicios, herramientas o librerías (Figura I.I B).
- **Stacks (Pilas):** Agrupaciones de paquetes complementarios entre sí que forman una librería de alto nivel (Figura I.I A).

A B

Fig. I.I. Descripción gráfica de Stacks y Package en ROS A) Pilas o stack. B) Paquetes o Package

Ambos poseen archivos de información detallando el contenido, la funcionalidad y los paquetes de los que depende. En las pilas aparece como *stack.xml* y en los paquetes como

manifest.xml. Las pilas se agrupan en repositorios equivalentes a los repositorios de Linux mediante la línea de comandos o el administrador de descargas.

Para comprender cómo se organizan los paquetes en las aplicaciones, conviene explicar primero los tipos de carpetas, con su contenido, y archivos que se pueden encontrar en un paquete ROS:

- **Carpeta bin:** ejecutables del paquete, lanzados como nodos.
- **Carpeta build:** archivos de compilación internos del paquete.
- **Carpeta src:** códigos de los programas (.cpp), que al compilarlos generarán los ejecutables de la carpeta bin.
- **Carpeta include:** archivos cabeceras (.h) de los programas.
- **Carpeta lib:** archivos de librerías (.lib).
- **Carpeta launch:** archivos de lanzamiento de aplicaciones completas (.launch).
- **Carpeta yaml:** archivos de la lista de parámetros
- **Carpeta msg:** tipos de mensajes (.msg) definidos en el paquete
- **Carpeta msg_gen:** archivos de cabeceras auto-generadas al compilar los tipos de mensajes definidos en el paquete.
- **Carpeta srv:** tipos de mensajes de los servicios definidos en el paquete.
- **Carpeta srv_gen:** archivos cabeceras auto-generadas al compilar los tipos de servicios definidos en el paquete.
- **Archivo CmakeLists.txt:** es una lista en la que se especifica al compilador qué debe compilar de los distintos paquetes: nodos, mensajes, servicios, etc.
- **Archivo manifest.xml:** contiene una descripción del paquete (función, autor, licencia, etc.) y una lista con los paquetes de los que depende.

b. Compilación

Para la compilación de un paquete en ROS se emplea CMake, es decir, una plataforma de código abierto que permite generar, compilar y comprobar los paquetes de software. Su uso es bastante sencillo, basta con que el paquete de ROS contenga un archivo llamado CMakeLists.txt que contiene una serie de marcos en función de lo que queramos crear y compilar. Los marcos más comunes que se utilizan son:

- **Rosbuild_add_executable (ejecutable src/programa.cpp):** crea en la carpeta bin el ejecutable del .cpp descrito en src.

- **Rosbuild_add_library (librería src/programa.cpp):** crea en la carpeta lib la librería del .cpp descrito en src.
- **Rosbuild_genmsg ():** auto-genera las cabeceras y archivos necesarios de los tipos de mensajes definidos en la carpeta msg del paquete y los guardará en la carpeta denominada msg_gen.
- **Rosbuild_gensrv ():** auto-genera las cabeceras y archivos necesarios de los tipos de servicios definidos en la carpeta srv del paquete y los guardará en la carpeta denominada srv_gen.

c. Nodos

Un nodo (Fig. I.II) es un proceso individual dentro del sistema ROS encargado de realizar un cómputo, enviar y/o recibir información de otros nodos y usar y ofrecer servicios. Se puede añadir que son los ejecutables de los paquetes de ROS ejecutándose. Cada nodo posee un nombre único, que lo identifica y lo distingue de los demás nodos en ejecución.

Los nodos se pueden escribir en C++ o Phyton usando la librería de cliente que corresponde a cada lenguaje de programación, o bien **roscpp** o bien **rospy.** Hay que tener en cuenta que la comunicación entre nodos no depende del lenguaje en el que estén escritos, ya que funcionarán siempre y cuando se cumpla que envíen y reciban mensajes del mismo tipo.

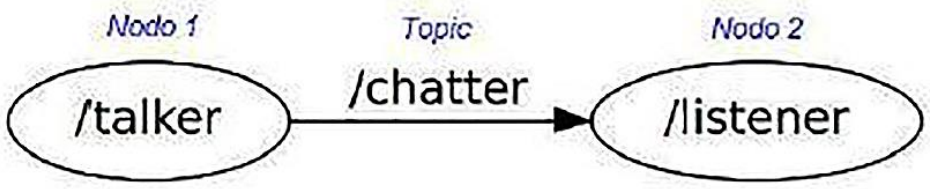


Fig. I.II. Descripción gráfica de Nodos

La librería **roscpp** es una implementación del lenguaje de programación C++ específica para ROS que contiene librerías, funciones y utilidades, además de las interfaces para utilizar los topics, servicios y parámetros de ROS para establecer una comunicación desde el programa con los demás nodos y aplicaciones que se estén ejecutando en el sistema.

Toda programación de nodos en ROS debe incluir las instrucciones siguientes (Fig. I.III):

```
#include "ros/ros.h"

int main(int argc, char **argv)
{
    ros::init(argc, argv, "nombre_nodo");
    ros::NodeHandle n;
              (...)

    return 0;
}
```

Fig. I.III. Instrucciones obligatorias para la creación de un nodo en ROS

- **"ros.h"** posee todas las cabeceras necesarias para el uso de las instrucciones más comunes del sistema de ROS, como las que hacen referencia al uso de topics y servicios, pero sin incluir los tipos de mensajes.
- **" ros::init (argc, argv, "nombre_nodo") "** inicializa el nodo en el sistema con el nombre que le asignemos y los argumentos que haya recibido al ejecutarlo.
- **"ros::NodeHandle n"** crea el principal punto de acceso a las comunicaciones de este nodo con el resto del sistema en ROS. Es una interfaz para crear suscripciones y publicaciones del nodo a los topics, así como para acceder al servidor de parámetros. Junto con init (), inicializa el nodo y lo elimina automáticamente al finalizar la ejecución del programa, mientras que "n" sería el identificado del proceso de este nodo y debe ser único.

A parte de esto, dentro de este apartado hay que indicar que **roscpp** permite ejecutar ciclos a una frecuencia concreta, ciclos a través de los que recibimos mensajes o controlamos un robot entre otras cosas (Fig. I.IV).

```
ros::Rate r(10);
while (ros::ok())
{
    (Código a ejecutar...)

r.sleep();
}
```

Figura I.IV. Instrucciones para ejecución de ciclos en ROS

- **"ros::Rate r(10)"** permite especificar la frecuencia a la que se va a ejecutar el bucle while(), siendo 10 la frecuencia en Hz.

- **"ros::ok()"** es una función que devuelve un valor falso cuando se produzca un ctrl + C, cuando haya sido expulsado el nodo porque el nombre ya existe o **"ros::shutdown()"** haya sido ejecutada.
- **"r.sleep ()"** es la función que asegura que se cumpla el ciclo.

d. Topics

Buses a través de los cuales los nodos envían o reciben mensajes. Para ello, el nodo debe comunicar al Master que desea publicar en él para enviar información o suscribirse en él para recibir su información. Existe la posibilidad de que varios nodos estén publicando y estén suscritos a la vez en un mismo topic, ya que los nodos publican o se suscriben a ellos pero no saben que otros nodos están publicando o suscritos a ese mismo topic (Fig. I.V).

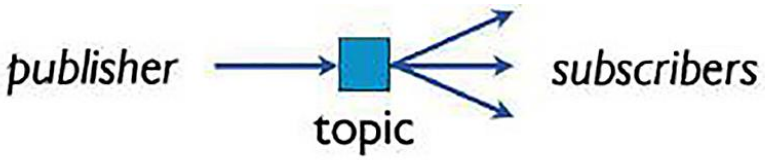


Fig. I.V. Descripción gráfica de un Topic

Los topics son unidireccionales, es decir, un nodo que manda información a través de un topic nunca recibirá respuesta a través de ese mismo topic, ni sabrá si sus mensajes están siendo recibidos. Para enviar una información y recibir una respuesta debe usarse el otro tipo de comunicación entre nodos, los servicios.

Por un topic se envían objetos de un tipo de mensaje definido en el mismo paquete o en alguno que depende del paquete en cuestión. La definición de estos objetos o mensajes se incluye en el programa mediante cabeceras generadas a partir de los archivos .msg.

Un topic viene identificado por su nombre, que debe ser único y normalmente es de la forma: "/nodo_que_lo_publica/nombre_topic". A la hora de su declaración, es necesario tanto el nombre como definir el tipo de menaje que se va a enviar por ellos y, por tanto, el tipo de mensaje que se va a recibir por ellos. El Master no asegura esta concordancia, por lo cual ha de tenerse especial cuidado en que al subscribirse a un topic se haga con el tipo de mensaje que se está enviando.

Las funciones básicas de **roscpp** para publicar un topic, y que se incluirán dentro de la función main (), son (Fig.I.VI):

```
#include "std_msgs/String.h"
       (...)

ros::Publisher chatter_pub = n.advertise<std_msgs::String
   >("chatter", 1000);
std_msgs::String msg;
msg.data = "hello world";
chatter_pub.publish(msg);
```

Figura I.VI. Instrucciones para Topics

- La primera instrucción incluye definiciones de la clase del tipo de mensajes que se va a utilizar, en este caso String del paquete std_msgs.
- La segunda instrucción crea el topic "**chatter**", con una cola de capacidad de 1000 mensajes y se define el tipo de mensaje que se va a enviar. Esta instrucción devolverá un objeto de la clase **"ros::Publisher**", denominado "**chatter_pub**" que sirve como identificador del topic dentro del programa y posee la función **publish()** que es la que se usa en la última instrucción para enviar el mensaje por el topic desde el programa o nodo.
- La tercera es simplemente un objeto que tendrá la estructura del tipo de mensaje.

La suscripción a un topic es equivalente (Fig. I.VII):

```
#include "std_msgs/String.h"
       (...)

void chatter_funcion(const std_msgs::String msg)
{
   ROS_INFO("Recibida cadena: [%s]", msg.data.c_str());
}

int main(int argc, char **argv)
{
   ros::Subscriber sub = n.subscribe("chatter", 1000,
      chatter_funcion);
          (...)
}
```

Figura I.VII. Suscripción a un Topic

- Con la última instrucción el nodo se suscribe al topic "**chatter"** y se declara que, al recibirse un mensaje por el topic se llamará a la función "**chatter_funcion**" definida en el mismo programa y que, como puede observarse, recibe como argumento el mensaje que llega por el topic.
- La función ROS_INFO () es equivalente al printf ().

La suscripción al topic no provoca que se reciban los mensajes. Para ello, ha de añadirse otra función en el punto del programa donde deseamos que se reciba.

- **ros:spin():** Esta función permite al programa mantenerse en un bucle en el que espera a que lleguen mensajes por el topic. Terminará una vez que ros::ok() devuelva un valor falso, es decir, una vez llamada a ros::shutdowm a través de un ctrl + C o porque el Master de ROS se ha apagado.
- **ros::spinOnce():** Se recibirán todos los mensajes que estén en cola en todos los topics en este punto de la ejecución, una vez completado el programa seguirá ejecutándose normalmente. Típicamente esta función va dentro de un bucle while ().

e. Servicios

Los servicios en ROS son la forma en que se envía un mensaje (request) desde un nodo a otro y éste le responde con otro mensaje (response). Un nodo puede ofrecer un servicio (server) y cualquier otro nodo puede utilizarlo (client) llamándolo junto con un mensaje y esperando una respuesta de éste (Fig. I.VIII). El servicio ofrecido por un nodo es totalmente independiente de los clientes.

Vienen definidos mediante archivos. Srv, que definen el tipo de servicio que se va a ofrecer, esto es el tipo de mensaje que se va a recibir y el tipo de mensaje que se va a devolver. Se deben incluir las cabeceras auto-generadas por los archivos .srv que contendrán las definiciones de la clase del tipo servicio.

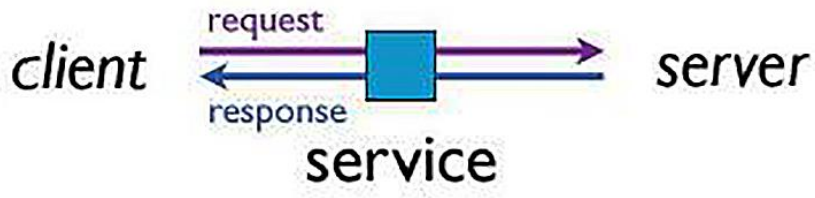


Figura I.VIII. Descripción gráfica de Servicio en ROS

Las funciones básicas que establece **roscpp** para generar y mantener un servicio son (Fig. I.IX):

```
#include "Matematicas/SumaDosEnteros.h"

bool fun_sum(Matematicas::SumaDosEnteros::Request &req,
   Matematicas::SumaDosEnteros::Response &res)
{
    res.sum = req.a + req.b;
    Return true;
}
int main(int argc, char **argv)
{
    (...)

    ros::ServiceServer service = n.advertiseService("
       suma_dos_enteros",fun_sum);
    (...)
}
```

Figura I.IX. Instrucciones para generar servicio en ROS

- La primera instrucción incluirá el tipo de servicio que se va a usar (*SumaDosEnteros*.srv) del paquete *Matemáticas*, que define la clase del tipo de servicio.
- La última instrucción crea propiamente el servicio, devolviendo su identificador dentro del programa. El nombre del servicio será *"suma_dos_enteros"* y la función que ejecutará el servicio al ser llamado será *"fun_sum"* que debe definirse dentro del programa. La función recibe como argumentos las direcciones donde se encuentra el mensaje recibido y donde debe escribir la respuesta.

De manera similar se crea un cliente a éste servicio (Fig. I.X):

```
int main(int argc, char **argv) {
    ros::ServiceClient client = n.serviceClient<Matematicas
       ::SumaDosEnteros>("suma_dos_enteros");

    beginner_tutorials::AddTwoInts srv;
    srv.request.a = 10;
    srv.request.b = -3;

    client.call(srv)
    return 0;
}
```

Figura I.X. Instrucciones para crear un cliente en ROS

- Lo primero que se hace es incluir el archivo de cabecera del tipo servicio.

- La segunda instrucción crea el cliente al servicio "*suma_dos_enteros*" que ofrecería el otro nodo y, se le define también el tipo de servicio, obteniendo el identificador client para llamar al servicio.
- La tercera instrucción sería un objeto de la clase del servicio que contiene los miembros request y response.
- A continuación, se rellena el objeto y se envía usando la función "call ()".

f. Mensajes

Estructuras de datos equivalentes a una estructura de C++, que se definen en la carpeta msg de los paquetes mediante archivos de texto de extensión .msg en forma de lista tipo-nombre (type1 name1; type2 name2).

La estructura puede estar formada por primitivas estándares equivalentes a las de C++ definidas en el paquete std_msgs o por otros tipos de mensajes (.msg) estándares o ya definidos en std_msgs o en cualquier otro paquete de ROS siempre que exista dependencia entre paquetes.

Los servicios definían sus tipos en los archivos .srv, los cuales son equivalentes a los .msg salvo que se componen de dos listas separadas por "-", primero con la lista de la estructura de request y segundo con la de response. Es posible crear variables de tipos de mensajes o servicios dentro del programa incluyendo los archivos de encabezamiento generados por estos y tratándolos igual que si fueran variables o estructura u objeto.

g. Master o roscore

Es un nodo que funciona de núcleo del sistema, proporcionando una plataforma mediante el registro de nombres, servicios y parámetros, lo cual hace posible la comunicación y envío de datos entre los diferentes nodos individuales del sistema. Sino lanzamos el Master es imposible la comunicación entre nodos (Fig. I.XI).

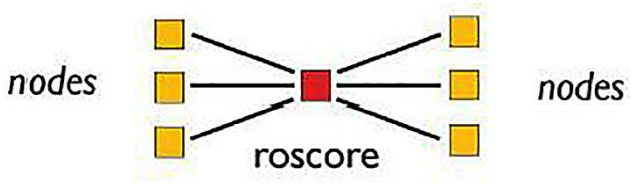


Figura I.XI. Descripción gráfica de Roscore

El roscore posee tres funciones como herramienta de ROS:

- ***Plataforma de comunicación para los nodos:*** Todos los nodos se dirigen antes al Master para crear y buscar el topic, antes de iniciar la comunicación entre ellos
- ***Servidor de Parámetros:*** Utilizado por los nodos para almacenar o leer parámetros en tiempo de ejecución, haciendo además posible que el usuario pueda ver y modificar estos parámetros por el terminal en el tiempo de ejecución.
- ***Registro de salida al que todos los nodos envían información*** **(/rosout)**

Para ejecutar el Master basta con lanzar a través del terminal: **$ roscore**

h. Lanzadores

En ROS existe una herramienta denominada **roslaunch** que permite lanzar múltiples nodos con sus argumentos, establecer una serie de parámetros en el servidor y otras opciones simplemente lanzando desde una terminal:

$ roslaunch nombre_paquete nombre_archivo.launch

Los lanzadores son archivos de configuración escritos en XML en los que se especifica los nodos a lanzar, los argumentos de entrada a éstos y los parámetros necesarios para crear una aplicación completa. Al ser un archivo XML, posee una serie de etiquetas que tienen una serie de opciones a rellenar. Las etiquetas más comunes y su utilidad se describen a continuación:

- **<launch>:** Primer elemento de cualquier archivo .launch. Es un identificador del tipo de archivo del que se trata. Se encontrará al principio y final en formato de cierre (/launch).
- <**node>:** Se emplea para lanzar un nodo. El nodo es una copia del nodo "tipo" que se quiere lanzar por lo que hay que darle un nombre diferente y especificar el paquete y el nombre del nodo "tipo".
- <**include**>:Permite incluir otro archivo launch a partir de su ruta
- **<param>:** Permite definir un parámetro en el Servidor de Parámetros.
- **<arg>:** Permite recibir argumentos al ejecutar el launcher y definirle unos valores por defecto.

i. Herramientas de línea de comandos

ROS como Meta-Sistema Operativo posee herramientas de línea de comandos:

- **$ roscore:** ejecuta el Master, el Servidor de Parámetros y el nodo de registro.
- **$ rosstack:** obtener información sobre una pila.
- **$rospack:** obtener información sobre un paquete.
- **$roscd:** acceder a un paquete en cuestión.
- **$rosls:** ver contenido del paquete en cuestión.
- **$roscreate-pkg:** crear un paquete en el lugar donde se encuentra, con la posibilidad de incluir las dependencias hacia otros paquetes.
- **$rosmake:** compilar el paquete deseado y todos los que dependan de él.
- **$rosdep:** descargar e instalar los paquetes de los que dependa el paquete en cuestión.
- **$rosrun:** ejecutar un nodo individual.
- **$rosnode:** permite obtener la lista de nodos en ejecución, información sobre un nodo o eliminarlo de la ejecución.
- **$rostopic:** obtener la lista de topics de los nodos en ejecución, información sobre uno o leer o publicar en él.
- **$rosservice:** permite obtener la lista de servicios de nodos en ejecución, información sobre uno o utilizarlo.
- **$rosmsg:** obtener información de un tipo de mensaje.

j. Rviz

Rviz es un visualizador 3D (Figura I.XII) que permite la observación de los datos de los sensores y la información del estado de ROS. Usando Rviz se puede visualizar la configuración actual de un modelo virtual de robot, se pueden mostrar representaciones en vivo de los valores de los sensores que se generan sobre Topics de ROS, datos de la cámara, etc. Es una herramienta de ROS que se emplea a lo largo del proyecto y con la que se observará el funcionamiento de las aplicaciones implementadas a lo largo del mismo.

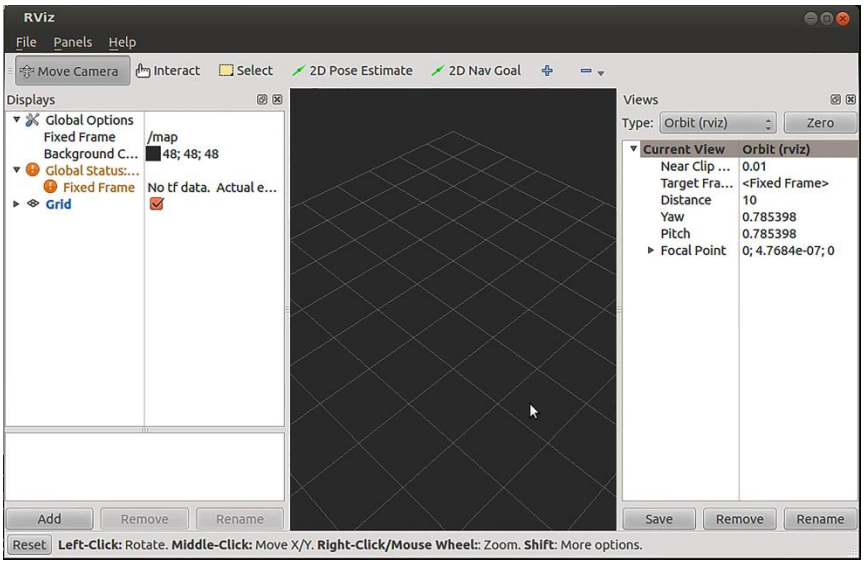


Fig. I.XII. Ventana Rviz

k. Rqt_graph

El paquete Rqt_graph tiene la función de crear gráficos dinámicos del sistema sobre lo que va a suceder en tiempo real. Permite saber cómo trabaja el sistema, y si lo está haciendo de forma correcta, ya que mostrará de forma gráfica los nodos que están en funcionamientos, sus dependencias y su forma de trabajar (Fig. I.XIII).

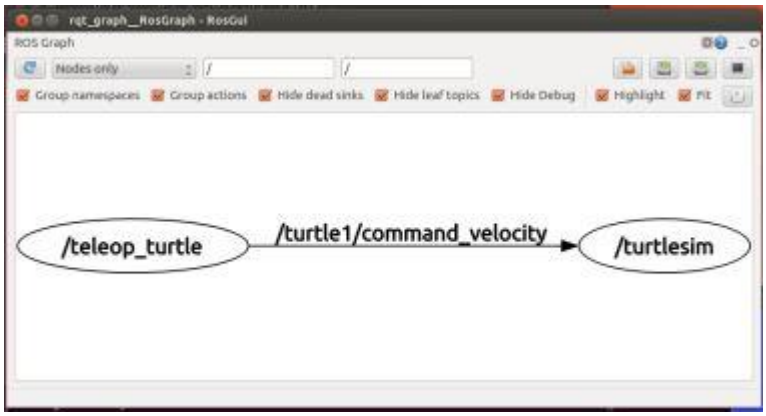


Fig. I.XIII. Ejemplo de Rqt_graph

Por las características descritas y, por la capacidad de ROS para operar en casos de fusión sensorial, el vehículo iCab trabaja con una arquitectura basada en ROS que permite simplificar la tarea de adquisición de datos global y la sincronización del sensor. Además, esta

arquitectura ha sido utilizada para generar la cartografía y la localización en navegación autónoma logrando resultados superiores a los de otros algoritmos [22] [23].

En el presente proyecto se emplea la plataforma ROS para desarrollar aplicaciones que permitan obtener un mapa del entorno cercano al vehículo a partir de la información captada por un sistema de visión estereoscópica, y posteriormente transformada en una nube de puntos o imagen 3D.

ANEXO II – CÓDIGO POINTCLOUD_NODE

En el presente anexo se adjunta el código implementado para transformar el mapa de disparidad en nube de puntos. Este código se encuentra en la carpeta src del paquete *Pointcloud,* es modificado a través de la herramienta Qt Creator, y se ejecuta a través del terminal de Linux empleando herramientas de la línea de comandos de ROS.

```
#include <ros/ros.h>
#include "ros/package.h"
#include <cv_bridge/cv_bridge.h>
#include <image_transport/image_transport.h>
#include <sensor_msgs/image_encodings.h>
#include <opencv2/imgproc/imgproc.hpp>
#include <opencv2/highgui/highgui.hpp>
#include <opencv2/calib3d/calib3d.hpp>
#include <message_filters/subscriber.h>
#include <message_filters/time_synchronizer.h>

#include <pcl/io/pcd_io.h>
#include <pcl_ros/point_cloud.h>
#include <pcl/point_types.h>
#include <pcl/console/parse.h>
#include <pcl_conversions/pcl_conversions.h>

#include <sstream>

using namespace std;
using namespace cv;
using namespace sensor_msgs;
using namespace std_msgs;
using namespace message_filters;

typedef pcl::PointCloud<pcl::PointXYZRGB> PointCloud;

ros::Publisher pub_pointcloud;

void pointCloudCallback(const ImageConstPtr& disp, const ImageConstPtr& l_image){

       //Conversión de Imágenes de mensaje en ROS a formato Open CV
    cv_bridge::CvImagePtr cv_disp_ptr, cv_left_ptr;
    try{
        cv_disp_ptr = cv_bridge::toCvCopy(disp, "mono8");
        cv_left_ptr = cv_bridge::toCvCopy(l_image, "mono8");
    }catch(cv_bridge::Exception& e){
        ROS_ERROR("Could not convert from image to 'mono8'.");
    }

    // Obtener Matriz de Reproyección Q
    double q[4][4] = {{ 1, 0, 0, -322.0614 }, { 0, 1, 0, -247.6637 }, { 0, 0, 0,
811.9104 }, { 0, 0, -1.0/0.119915, 0 }};
    Mat Q( 4, 4, CV_32F, q );
```

```
    // Generación de imágenes en formato Open CV
       //Imagen donde guardar los datos del mapa de disparidad
    static Mat disparity(cv_disp_ptr->image.rows, cv_disp_ptr->image.cols,
CV_8UC1);
    disparity = cv_disp_ptr->image;

       //Imagen donde guardar los datos de la imagen izquierda
    static Mat left(cv_left_ptr->image.rows, cv_left_ptr->image.cols, CV_8UC1);
    left = cv_left_ptr->image;

    // Transforma la imagen izquierda en escala de grises a color
    static Mat bgr_image;
    cvtColor(left, bgr_image, CV_GRAY2BGR);

    // Creación de una imagen con el doble de precisión para el mapa de
disparidad
    static Mat disp32F;
    disparity.convertTo(disp32F, CV_32F, 1./16);

    // Construir la reproyección y la imagen donde guardar la nube de puntos
    static Mat XYZ = Mat::zeros(left.size(), CV_32FC3);
    reprojectImageTo3D( disparity, XYZ, Q, true);

    // Generación de la nube de puntos
    PointCloud::Ptr point_cloud_ptr (new pcl::PointCloud<pcl::PointXYZRGB>);

    double px, py, pz;
    uchar pr, pg, pb;

    for (int i = 0; i < bgr_image.rows; i++)
    {
        uchar* rgb_ptr = bgr_image.ptr<uchar>(i);
        uchar* disp_ptr = disparity.ptr<uchar>(i);
        double* recons_ptr = XYZ.ptr<double>(i);
        for (int j = 0; j < bgr_image.cols; j++)
        {
            //Obtener coordenadas 3D

             uchar d = disp_ptr[j];
             if ( d == 0 ) continue; //Discard bad pixels
             double pw = -1.0 * static_cast<double>(d) * (-1/0.119915) + 0;
             px = static_cast<double>(j)  - 322.0614; //-cx
             py = static_cast<double>(i) - 247.6637; //-cy
             pz = 811.9104; //focal length

             // Normalizar los puntos
             px = px/pw;
             py = py/pw;
             pz = pz/pw;

             //Ontener información RGB
             pb = rgb_ptr[3*j];
             pg = rgb_ptr[3*j+1];
             pr = rgb_ptr[3*j+2];

             //Insertar información en la estructura de la nube de puntos
             pcl::PointXYZRGB point;
             point.x = pz;
             point.y = -px;
             point.z = -py;
```

```
                uint32_t rgb = (static_cast<uint32_t>(pr) << 16 |
                static_cast<uint32_t>(pg) << 8 | static_cast<uint32_t>(pb));
                point.rgb = *reinterpret_cast<float*>(&rgb);
                point_cloud_ptr->points.push_back (point);
            }
        }

        point_cloud_ptr->width = (int) point_cloud_ptr->points.size();
        point_cloud_ptr->height = 1;
        point_cloud_ptr->header.frame_id = disp->header.frame_id;
        point_cloud_ptr->header.stamp = pcl_conversions::toPCL(disp->header.stamp);

        //Conversión de datos a mensajes de ROS para su publicación
        sensor_msgs::PointCloud2 ros_pointcloud;
        pcl::toROSMsg(*point_cloud_ptr, ros_pointcloud);
        pub_pointcloud.publish(ros_pointcloud);
}

int main(int argc, char **argv)
{
           //Inicialización del nodo
        ros::init(argc, argv, "pointcloud_node");
        ros::NodeHandle nh;

           //Creación de la publicación del mensaje
        pub_pointcloud =
nh.advertise<sensor_msgs::PointCloud2>("stereo_camera/pointcloud",1);
        pub_pointcloud = nh.advertise<sensor_msgs::PointCloud2>("cloud_in",1);

           //Subcripción a topics
        message_filters::Subscriber<Image> disparity_sub(nh,
"stereo_camera/disparity", 1);
        message_filters::Subscriber<Image> l_image_sub(nh,
"stereo_camera/left/image_rect", 1);
        TimeSynchronizer<Image, Image> sync(disparity_sub, l_image_sub, 10);
        sync.registerCallback(boost::bind(&pointCloudCallback, _1, _2));

        ros::spin();
        destroyAllWindows();
        return 0;
}
```

ANEXO III - ARCHIVO DE LANZAMIENTO POINTCLOUD_TO_LASERSCAN_SAMPLE_NODE

En este caso en lugar de aportar el código referente al paquete, se incluye el archivo de lanzamiento del mismo que recoge los distintos parámetros que caracterizan a la aplicación, mostrando el valor con el que se ha obtenido el resultado final de esta etapa.

Se ha decidido colocar el archivo .launch en lugar de todo el código del paquete debido a que el paquete se encuentra ya presente en la librería de ROS, y solo es necesario descargarlo a través del wiki de ROS para poder hacer uso del código y ejecutarlo. En este proyecto, solo se han realizado modificaciones en los parámetros del paquete, sin realizar ningún cambio en las demás líneas del algoritmo, por lo que se incluye el archivo de lanzamiento *simple_node.launch* que posee los parámetros principales que establecen las propiedades del Fake Laser y el valor que presenta cada uno de ellos al ejecutar el nodo.

```
<launch>

    <arg name="camera" default="camera" />

    <!-- start sensor-->

    <!-- run pointcloud_to_laserscan node -->
    <node pkg="pointcloud_to_laserscan"
type="pointcloud_to_laserscan_node"
name="pointcloud_to_laserscan">

        <remap from="cloud_in" to="cloud_in"/>
        <remap from="scan" to="$(arg camera)/scan"/>
      <remap from="scan" to="/scan"/>
        <rosparam>
            #target_frame: # Leave disabled to output scan in
pointcloud frame
            transform_tolerance: 0.01
            min_height: 0.0
            max_height: 1.0

            angle_min: -1.0472 # -M_PI/3
            angle_max: 1.0472 # M_PI/3
            angle_increment: 0.00581 # M_PI/540.0
            scan_time: 0.3333
            range_min: 0.45
            range_max: 10.0
            use_inf: true
```

```
            # Concurrency level, affects number of pointclouds
queued for processing and number of threads used
            # 0 : Detect number of cores
            # 1 : Single threaded
            # 2->inf : Parallelism level
            concurrency_level: 0
        </rosparam>

    </node>

</launch>
```

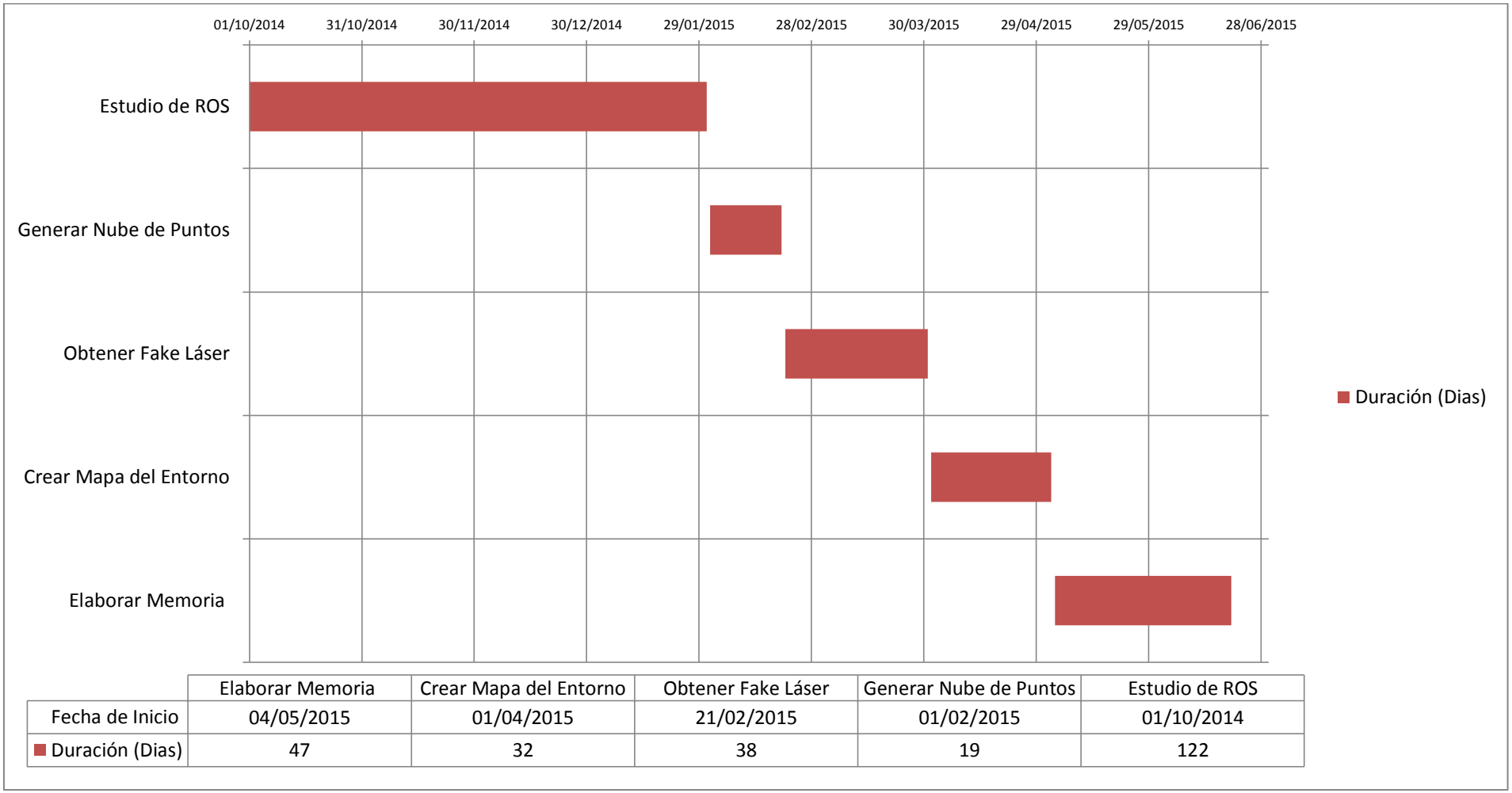
ANEXO IV - ARCHIVO DE LANZAMIENTO HECTOR_MAPPING_MAPPING_DEFAULT

En el presente anexo se está en la misma situación que en el anexo 2, por lo que las sentencias que se incluyen a continuación pertenecen al archivo de lanzamiento del nodo del paquete hector_mapping, llamado *mapping_defautl.launch.*

```
<launch>
  <arg name="tf_map_scanmatch_transform_frame_name"
default="scanmatcher_frame"/>
  <arg name="base_frame" default="base_footprint"/>
  <arg name="odom_frame" default="odom"/>
  <arg name="pub_map_odom_transform" default="false"/>
  <arg name="scan_subscriber_queue_size" default="50"/>
  <arg name="scan_topic" default="scan"/>
  <arg name="map_size" default="2048"/>

  <node pkg="hector_mapping" type="hector_mapping"
name="hector_mapping" output="screen">

    <param name="map_frame" value="map" />
    <param name="base_frame" value="$(arg base_frame)" />
    <param name="odom_frame" value="$(arg odom_frame)" />

    <!-- Tf use -->
    <param name="use_tf_scan_transformation" value="true"/>
    <param name="use_tf_pose_start_estimate" value="false"/>
    <param name="pub_map_odom_transform" value="$(arg
pub_map_odom_transform)"/>

    <!-- Map size / start point -->
    <param name="map_resolution" value="0.050"/>
    <param name="map_pub_period" value="2.0"/>
    <param name="laser_max_dist" value = "40.0" />
    <param name="laser_min_dist" value = "0.4" />
    <param name="map_size" value="$(arg map_size)"/>
    <param name="map_start_x" value="0.5"/>
    <param name="map_start_y" value="0.5" />
    <param name="map_multi_res_levels" value="2" />

    <!-- Map update parameters -->
    <param name="update_factor_free" value="0.4"/>
    <param name="update_factor_occupied" value="0.9" />
    <param name="map_update_distance_thresh" value="0.1"/>
    <param name="map_update_angle_thresh" value="0.06" />
    <param name="laser_z_min_value" value = "-1.0" />
    <param name="laser_z_max_value" value = "1.0" />

    <!-- Advertising config -->
```

```
    <param name="advertise_map_service" value="true"/>

    <param name="scan_subscriber_queue_size" value="$(arg
scan_subscriber_queue_size)"/>
    <param name="scan_topic" value="$(arg scan_topic)"/>

    <!-- Debug parameters -->
    <!--
      <param name="output_timing" value="false"/>
      <param name="pub_drawings" value="true"/>
      <param name="pub_debug_output" value="true"/>
    -->
    <param name="tf_map_scanmatch_transform_frame_name"
value="$(arg tf_map_scanmatch_transform_frame_name)" />
  </node>
    <!--
  <node pkg="tf" type="static_transform_publisher"
name="map_nav_broadcaster" args="0 0 0 0 0 0 map nav 100"/>
  -->
</launch>
```

ANEXO V - DIAGRAMA DE GANTT



	Elaborar Memoria	Crear Mapa del Entorno	Obtener Fake Láser	Generar Nube de Puntos	Estudio de ROS
Fecha de Inicio	04/05/2015	01/04/2015	21/02/2015	01/02/2015	01/10/2014
Duración (Dias)	47	32	38	19	122